

## 1. Code Composer Studio Projects

1. [Code Composer Studio v4 DSP/BIOS Project](#)
2. [Code Composer Studio v4 DSP/BIOS and C6713 DSK](#)
3. [Creating a TI Code Composer Studio Simulator Project](#)
4. [Code Composer Studio v3.3 DSP/BIOS and C6713 DSK](#)

## 2. Lab 1

1. [TI DSP/BIOS LOG Module](#)
2. [LOG Lab](#)

## 3. Lab 2

1. [TI DSP/BIOS TSK Module](#)
2. [TI DSP/BIOS SEM Module](#)
3. [TSK SEM Lab](#)

## 4. Lab 3

1. [TI DSP/BIOS QUE Module](#)
2. [QUE Lab](#)

## 5. Lab 4

1. [Microsoft Visual Basic Tasks and Semaphores](#)
2. [Visual Basic Threads and Semaphores Lab](#)

## 6. Lab 5

1. [Microsoft Visual Basic Queues](#)
2. [Microsoft Visual Basic Delegates](#)
3. [Microsoft Visual Basic Queue Lab](#)
4. [Microsoft Visual Basic Delegate Lab](#)

## Code Composer Studio v4 DSP/BIOS Project

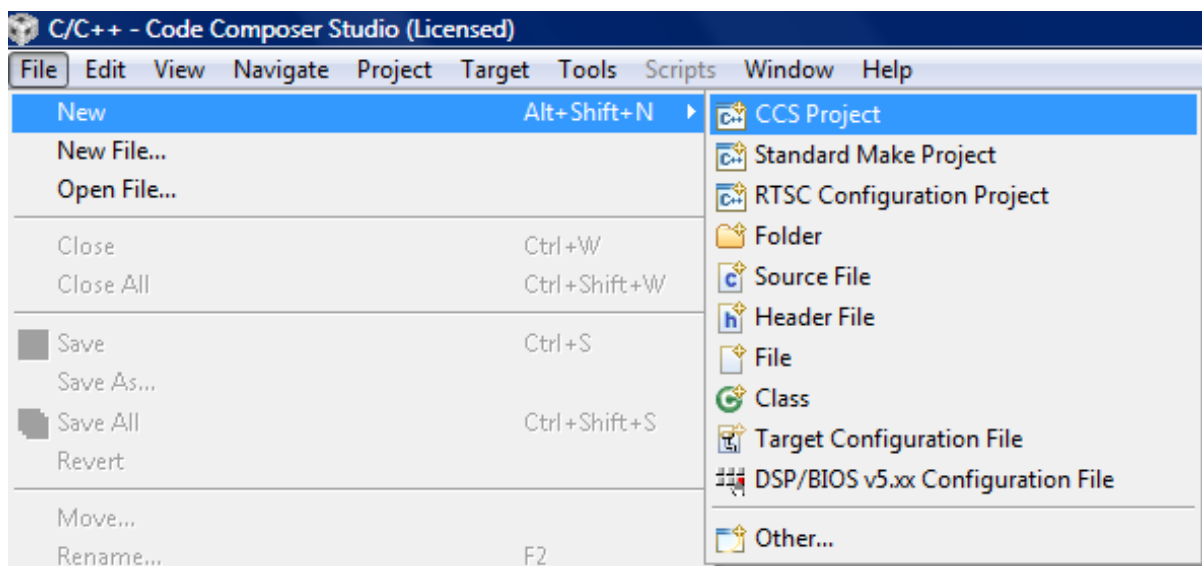
This module describes the process of creating, building and debugging a Code Composer Studio (CCS) v4 DSP/BIOS based program project.

### Introduction

This module describes how to create a Code Composer Studio (CCS) project that executes a simple DSP/BIOS based program. The target will be the TI simulator and the processor used is the TMS320C67xx.

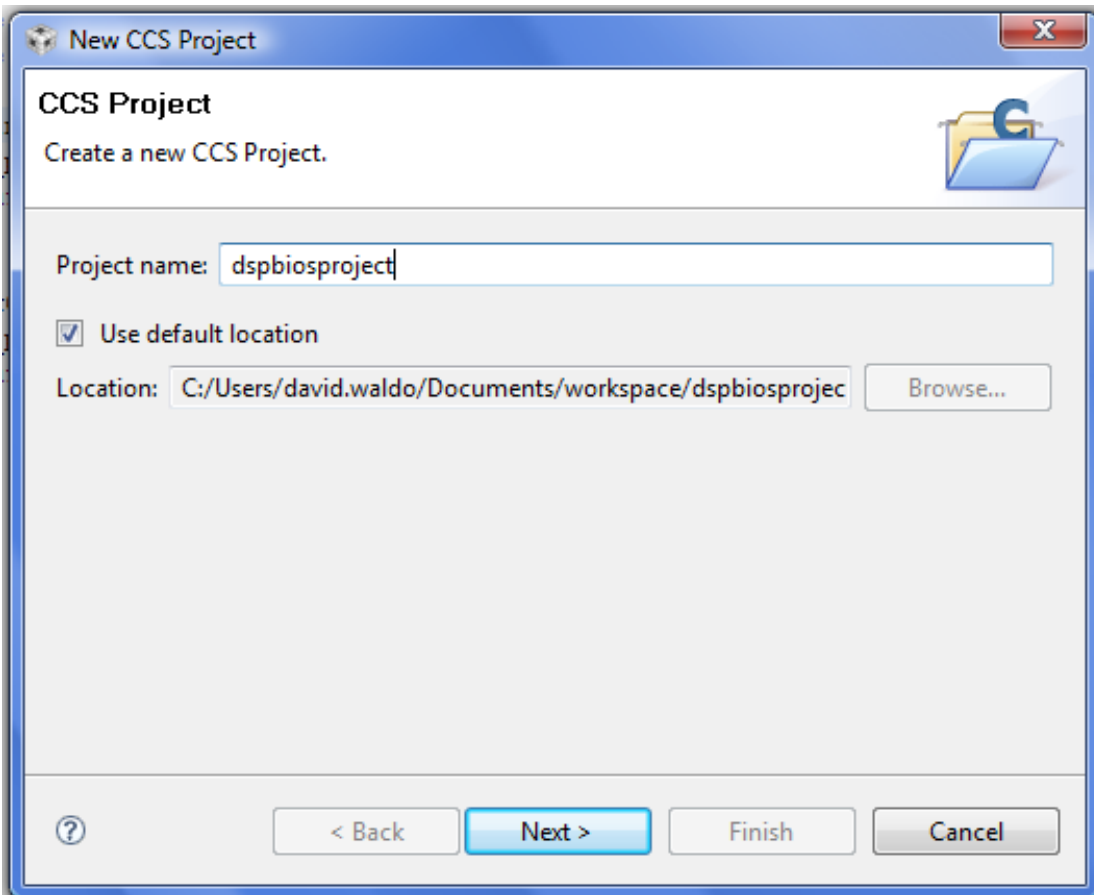
### Create a DSP/BIOS CCS project

To create a CCS project select **File->New->CCS Project**.



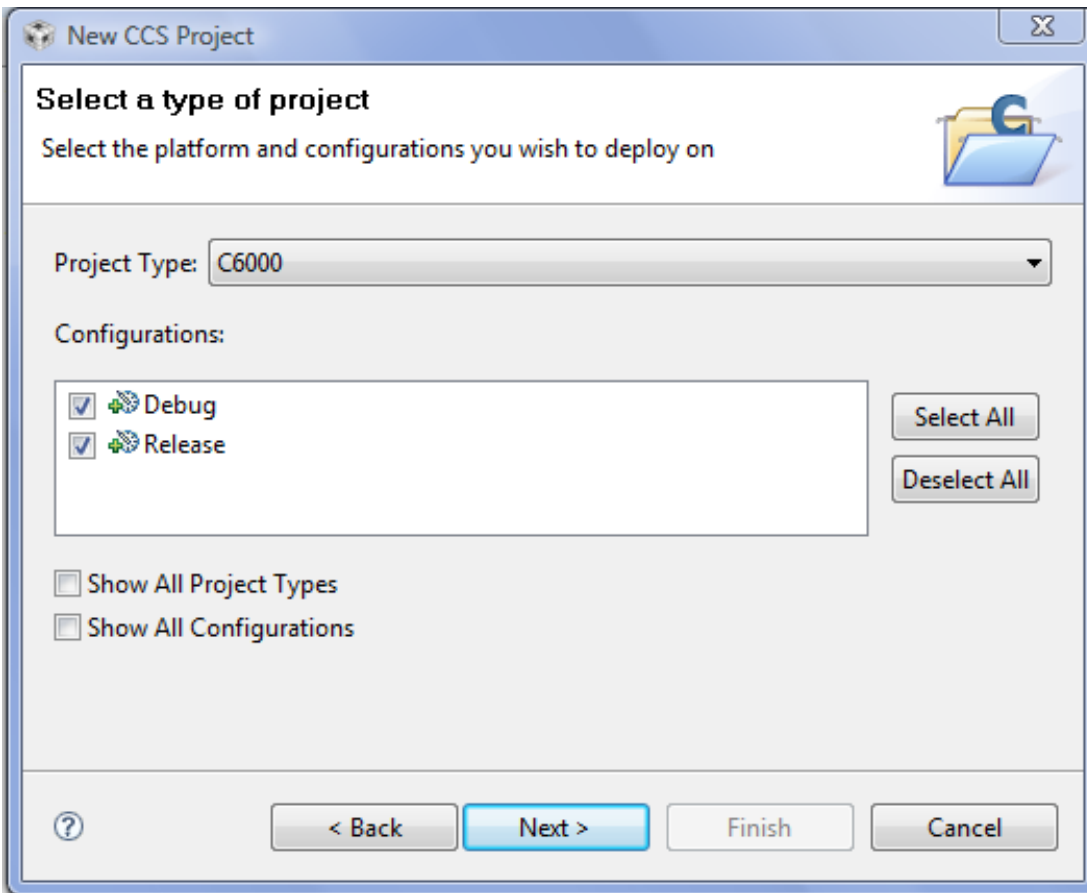
Screenshot of CCS Project menu

This will bring up a window where you can enter the name of the project. The location selected is the default location for project files. Press **Next**.



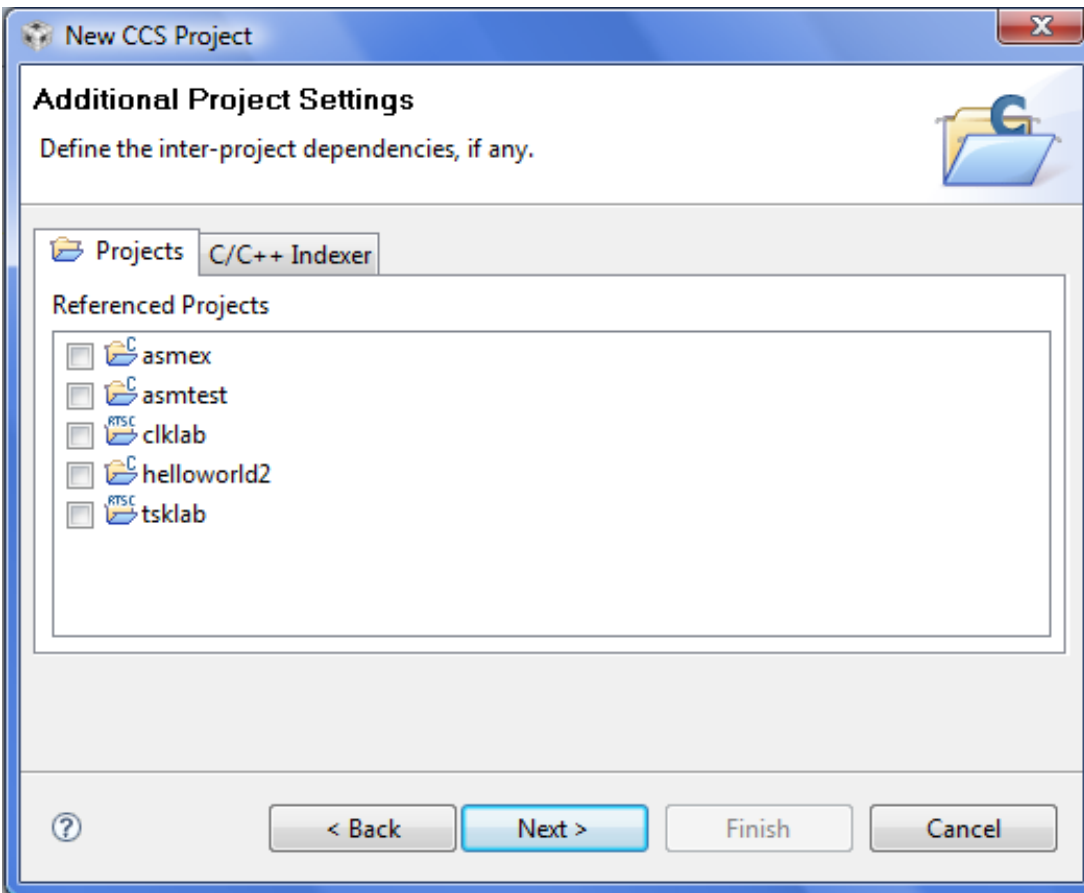
CCS Project name and location

Since the example uses the TMS320C67xx processor the project type selected is **C6000**. The project configurations are **Debug** and **Release**. Select the **Next** button.



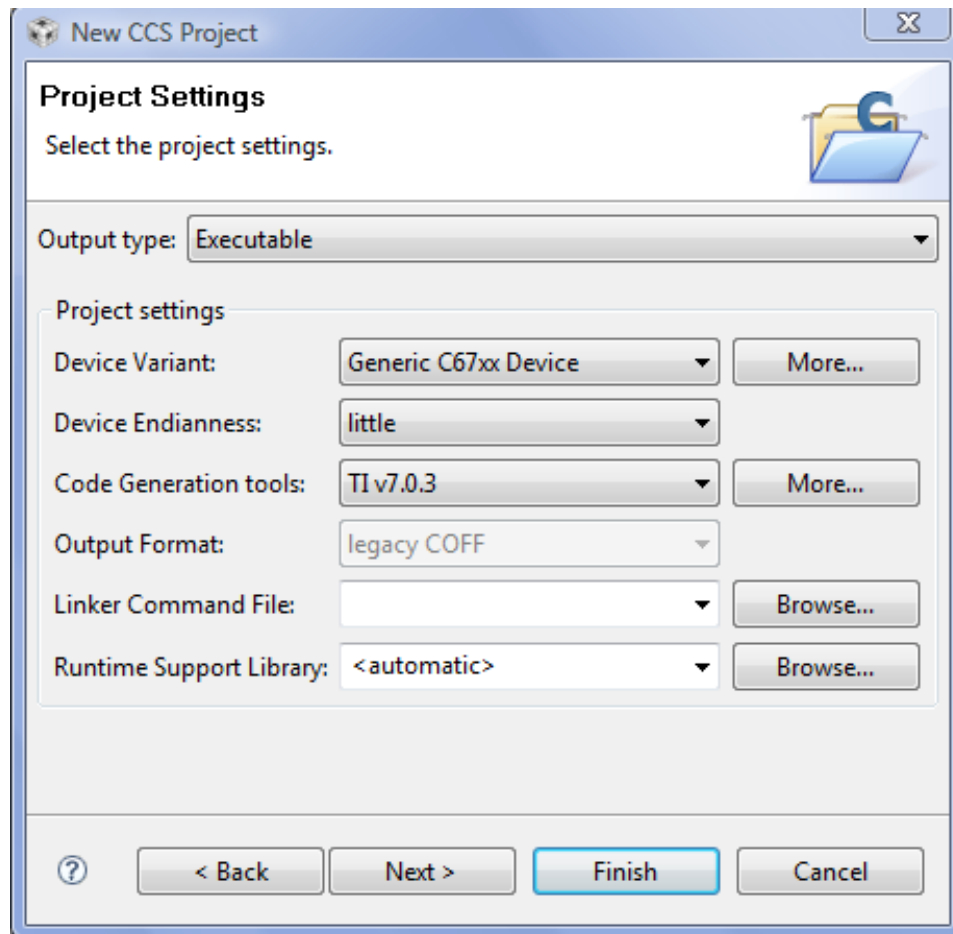
Type of CCS project

If there are any project dependencies they are selected on the next screen.  
Select the **Next** button.



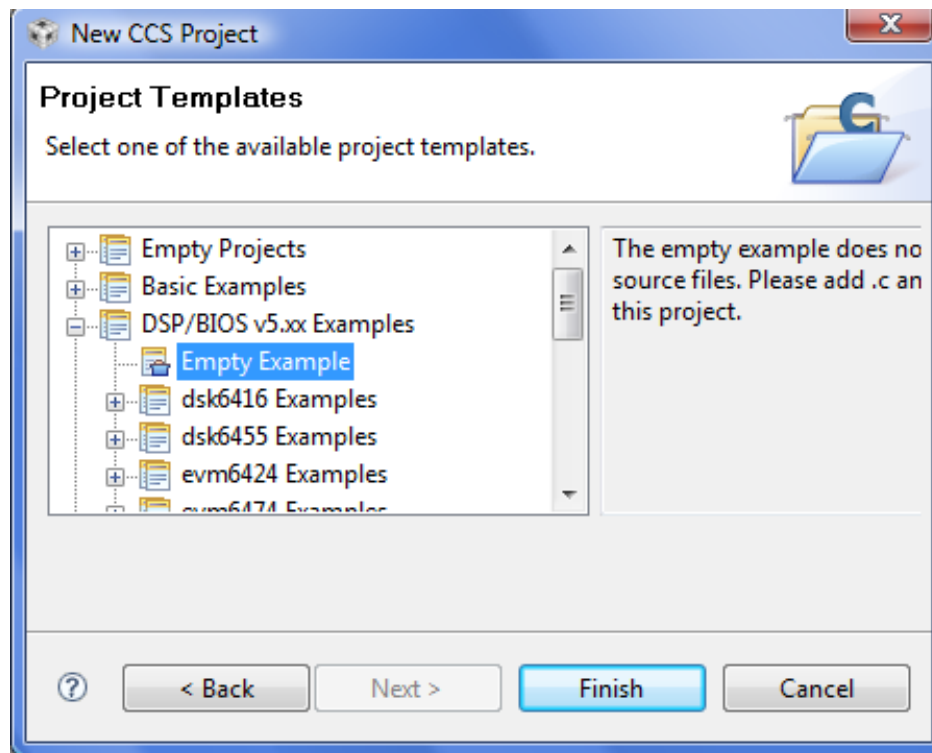
CCS Project dependencies

On the **Project Settings Screen**, select the items that pertain to the type of project to be created. Since the project will be executed select **Output Type: Executable**. The processor type is TMS320C67xx so the **Device Variant** is **Generic C67xx Device**. This project will use **Little Endian**. The code generation tools are the latest version (in this case TI v7.0.3). The runtime support library is selected to match the device variant and the endianness selected. The library can be selected automatically. Press **Next**.

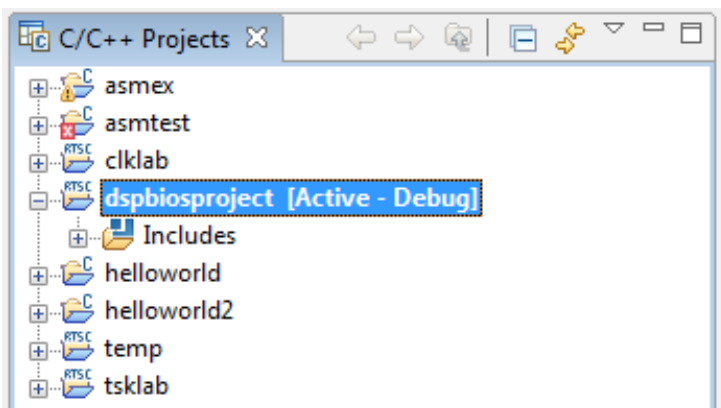


Project settings window

Since the project will be based on DSP/BIOS an empty DSP/BIOS example project. This will set up project parameters for DSP/BIOS.



After the projects settings have been set, select **Finish** and the project will show up in the **C/C++ Projects** view.

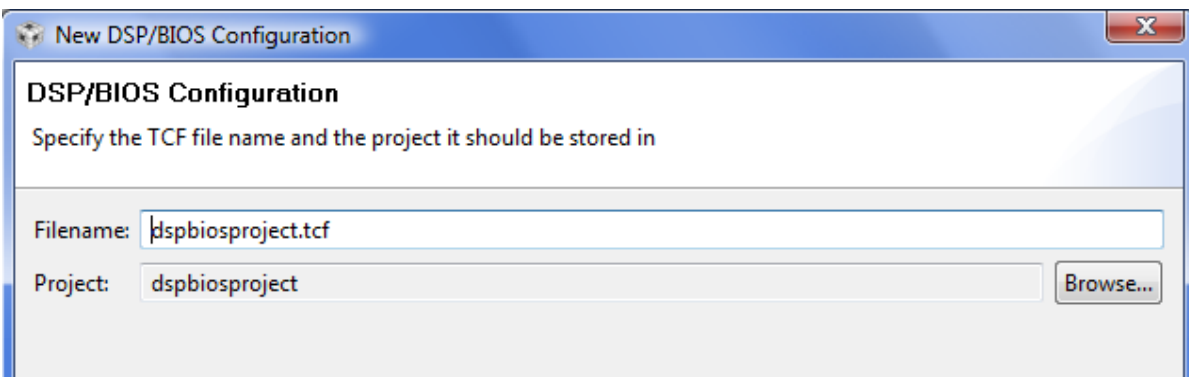


C/C++ projects view

## Add a DSP/BIOS configuration file

The easiest way to set up DSP/BIOS in a CCS project is to use the DSP/BIOS configuration tool. This generates a file with a .tcf extension that is included in the project. This file will generate code that gets included in the project for setting up DSP/BIOS. It also generates a header file that can be included in other C/C++ files. For the file name **dspbiosproject.tcf** the header file name will be **dspbiosprojectcfg.h**.

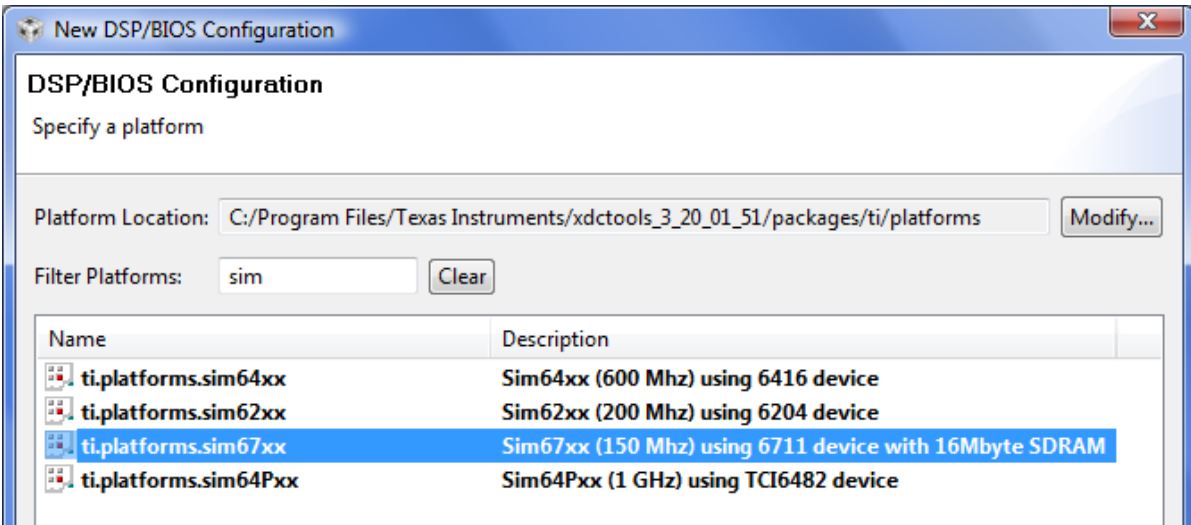
To create a DSP/BIOS configuration file, select **File->New->DSP/BIOS v5.xx Configuration File** (you may need to select **File->New->Other** first, then select **DSP/BIOS v5.xx Configuration File**). This will bring up a dialog where the name of the file is entered.



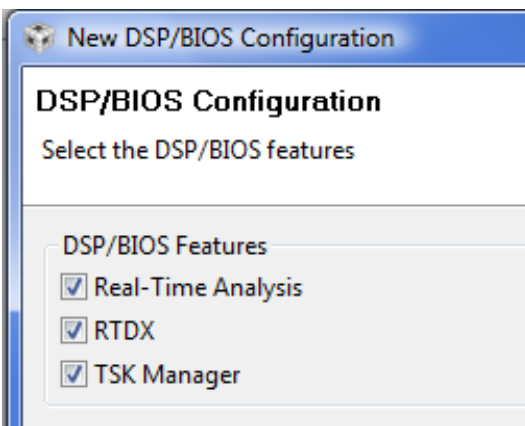
DSP/BIOS Configuration file name

There are default configuration files for different devices. For this example the C67xx simulator is used. The configuration will have 16Mbytes of SDRAM in the memory section. Select **Next** and a dialog for the DSP/BIOS features will come up.



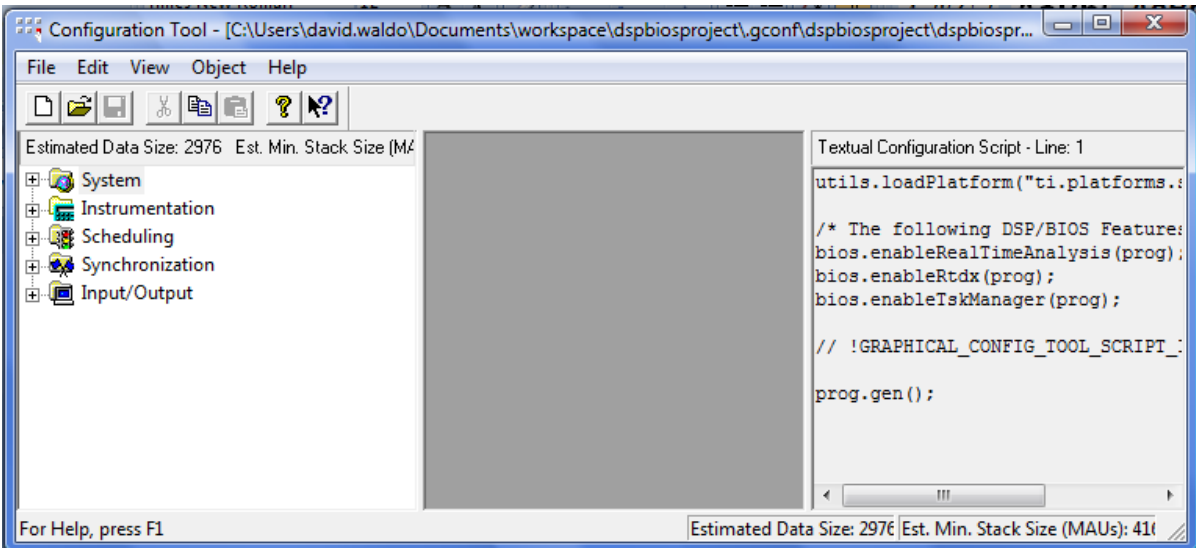


DSP/BIOS platform for default settings



DSP/BIOS features  
selection

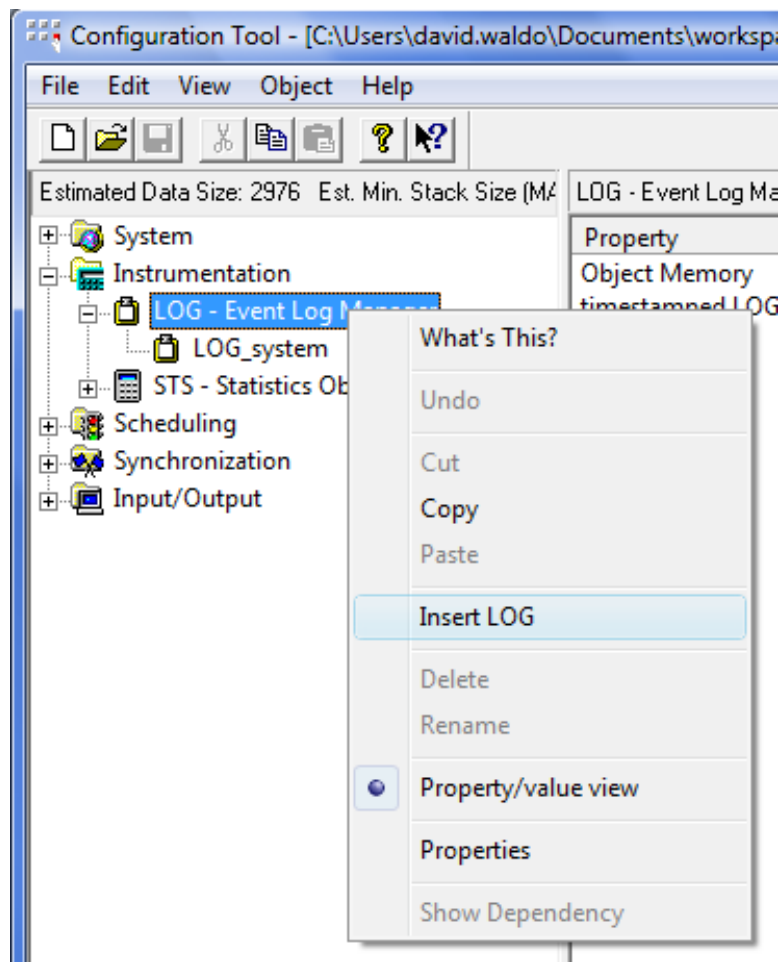
Select **Finish** and a new file is created with the default configuration and is added to the project and the configuration tool is opened.



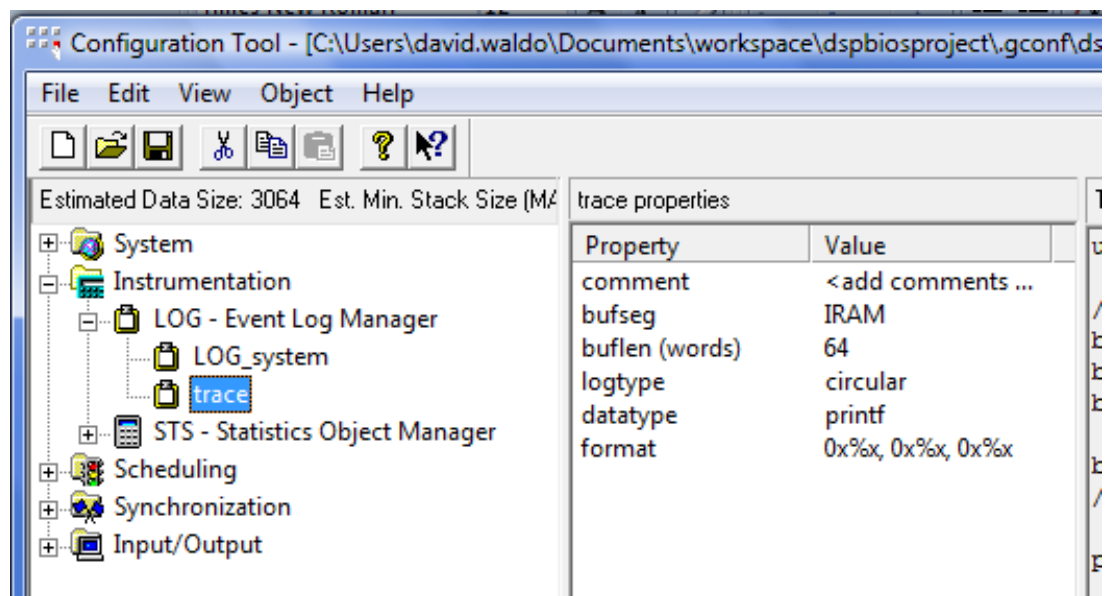
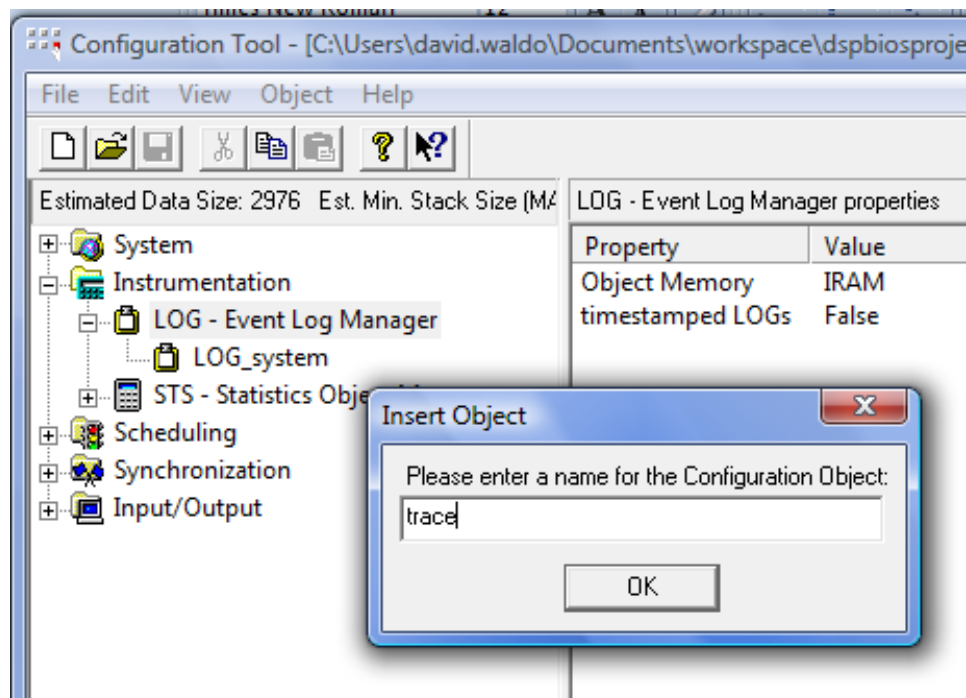
## DSP/BIOS configuration tool

In the configuration tool is where the DSP/BIOS objects are set up. In this example there are only two objects that will be set up. The first is a trace log that is common to many DSP/BIOS projects. The second is a task object that will be used to print to the log.

To set up the LOG object, expand the Instrumentation item and right click on the LOG item. Select Insert LOG and give the LOG object the name **trace**.

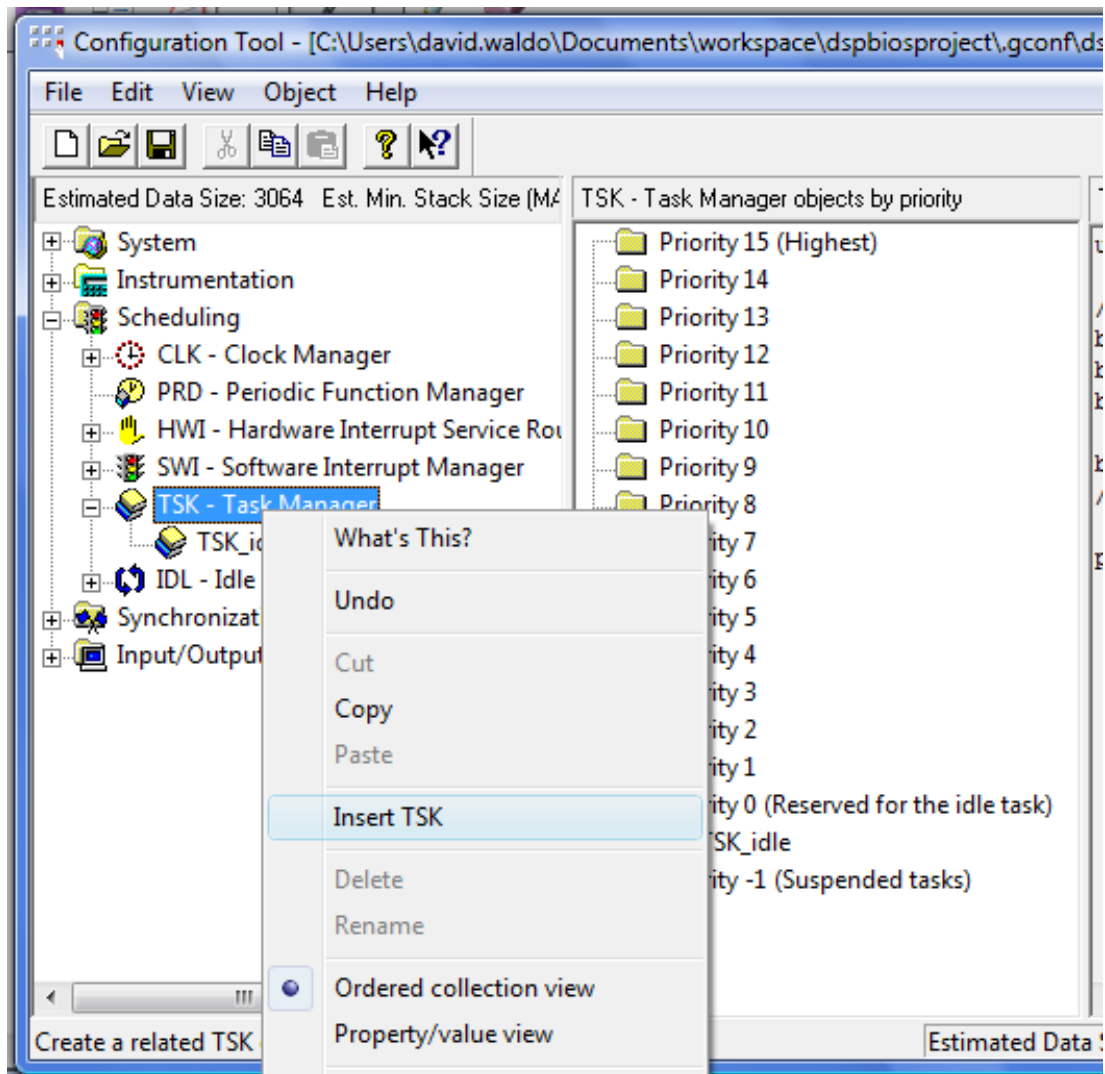


Insert LOG menu item

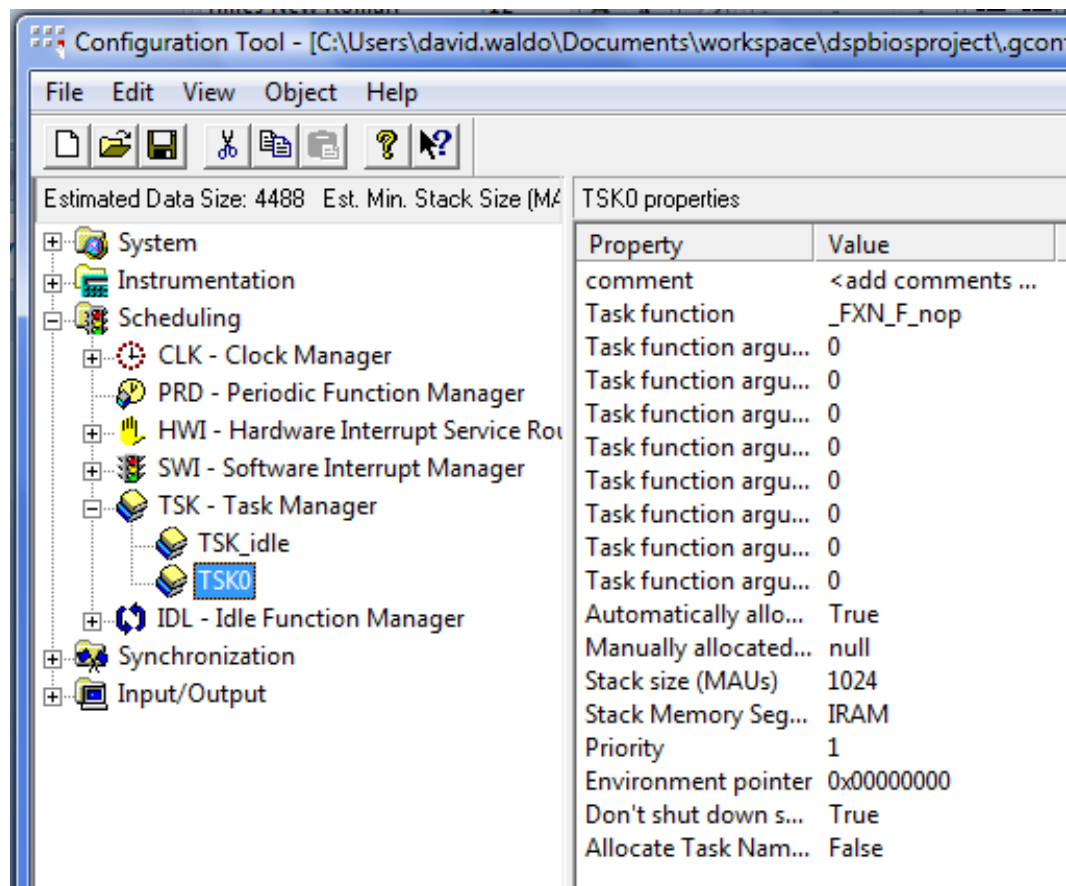


DSP/BIOS configuration tool showing the trace LOG object

Next insert a task (TSK) object and give it the name **TSK0**.



Inserting a TSK object



TSK object

In order for the TSK object to be associated with a function the Task function name must be changed. Right click on the **TSK0** object and select **Properties**. Under the Function tab, type **\_funTSK0**. Be sure to include the underscore at the beginning of the name. The C program function name will be **funTSK0** but the assembly code name will have an underscore at the beginning. Be sure to note that the name of the task object and the name of the function that it uses cannot be the same name. In this example the object name is TSK0 and the function name is **funTSK0**.

After the objects have been set up, save and close the configuration file.

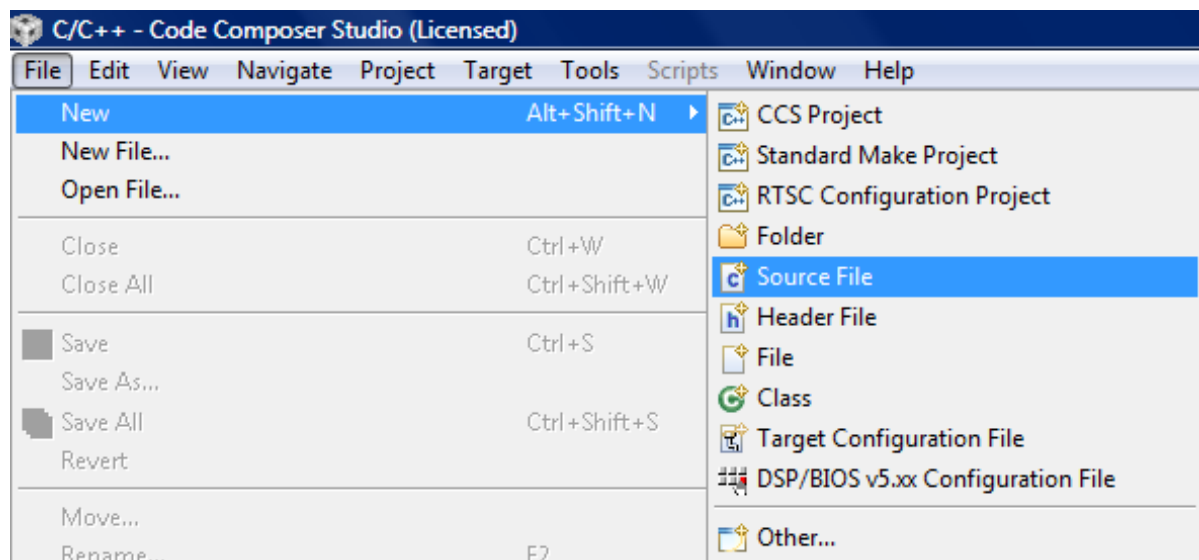
When your project gets built, the configuration file automatically generates files. One of these files is a header file that can be used in your programs. If

your configuration file is named `file.tcf` then the header file that gets made will be called `filecfg.h`. In the source files that use the DSP/BIOS object this header file can be included. Use a statement like:

```
#include "filecfg.h"
```

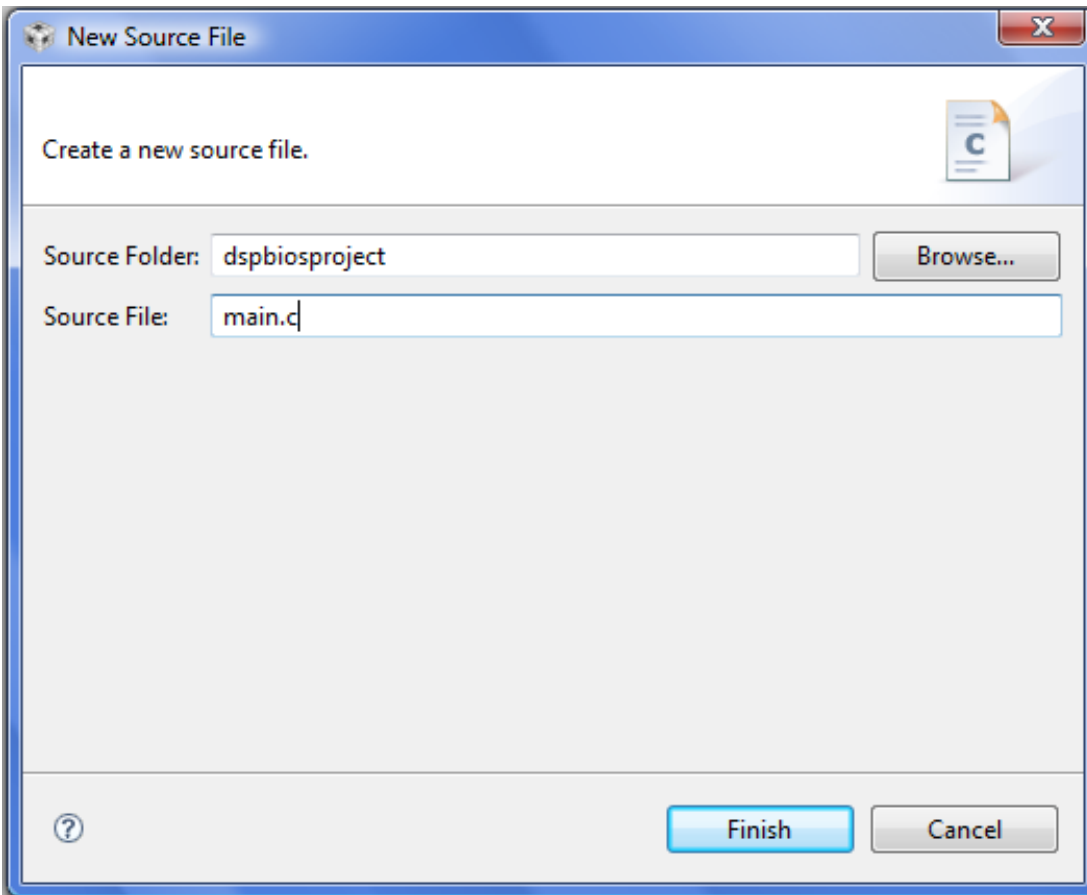
## Add code files

When using DSP/BIOS in a project, very little code is in the `main` function. Other objects are used like tasks and interrupts. In this example a simple task will be made that prints to a LOG buffer. Create a new C program file. Select **File->New->Source File**.



New source file

The **New Source File** dialog opens and you can enter the source file name. Add a `.c` extension for a C program file. The source folder should be the folder of your project. Select **Finish** and a new file is opened.



New source file dialog

Enter the C code. The file must contain a **main** function. After entering the code, save the file.

```
#include <std.h>
#include <log.h>
#include <tsk.h>
#include "dspbiosprojectcfg.h" //header
generated by dspbiosproject.tcf

void main(void){
LOG_printf(&trace,"In main function");
return;
}
```



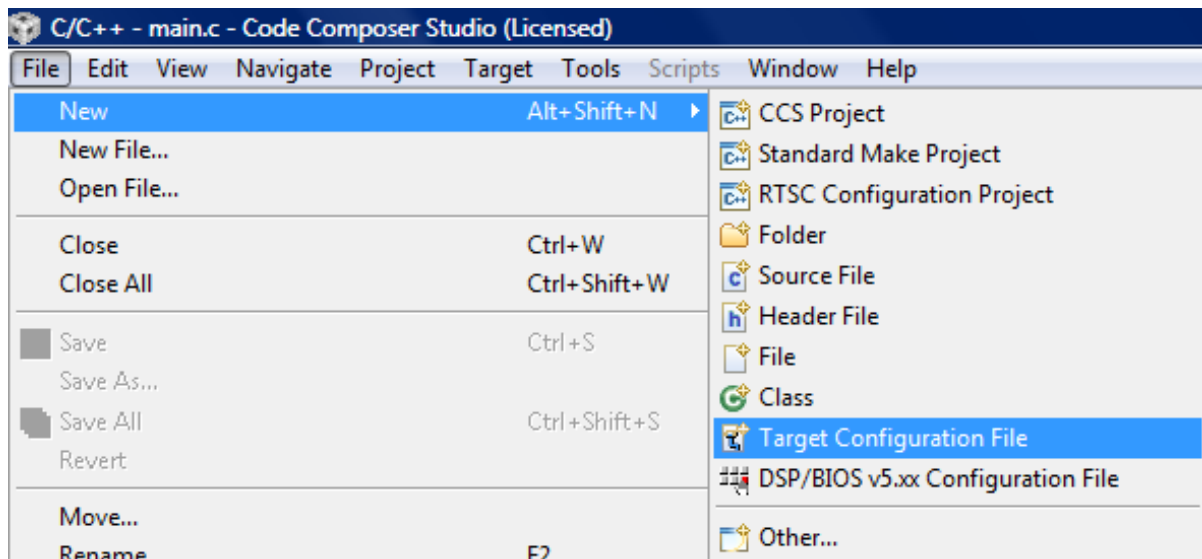
```
void funTSK0(void){
LOG_printf(&trace,"In TSK0");
return;
}
```

In the code above, note that the header file **dspbiosprojectcfg.h** is included. This is the header file generated by the DSP/BIOS configuration file. Also, the function **funTSK0** is the function that will be executed when TSK0 gets scheduled to run.

## Create a target configuration

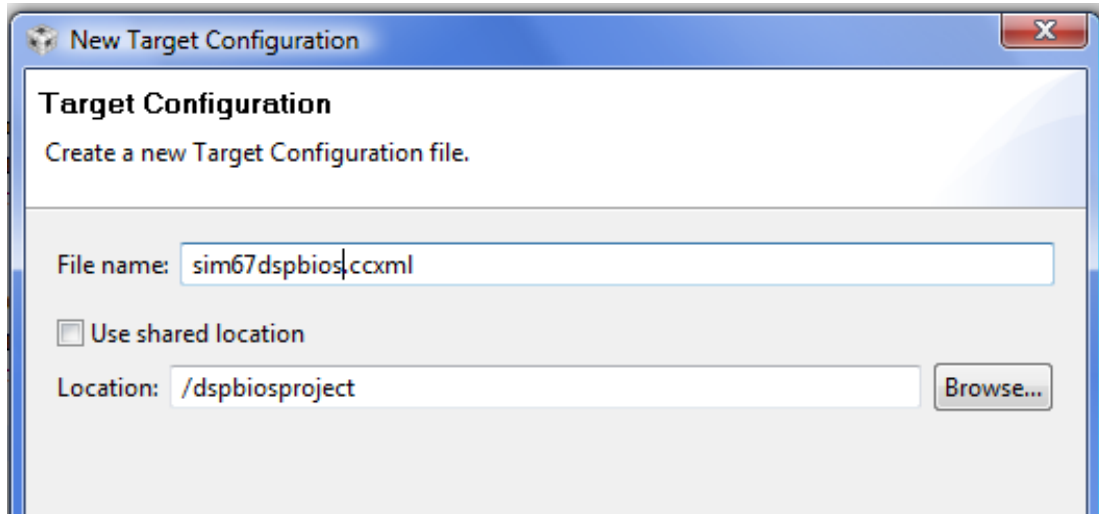
In order to build and run your project you need to have a target configuration. This file will contain information about the emulator that will be used and the target processor. In this module the target configuration will be the TI simulator for the C6713 processor.

First select **File->New->Target Configuration File**.



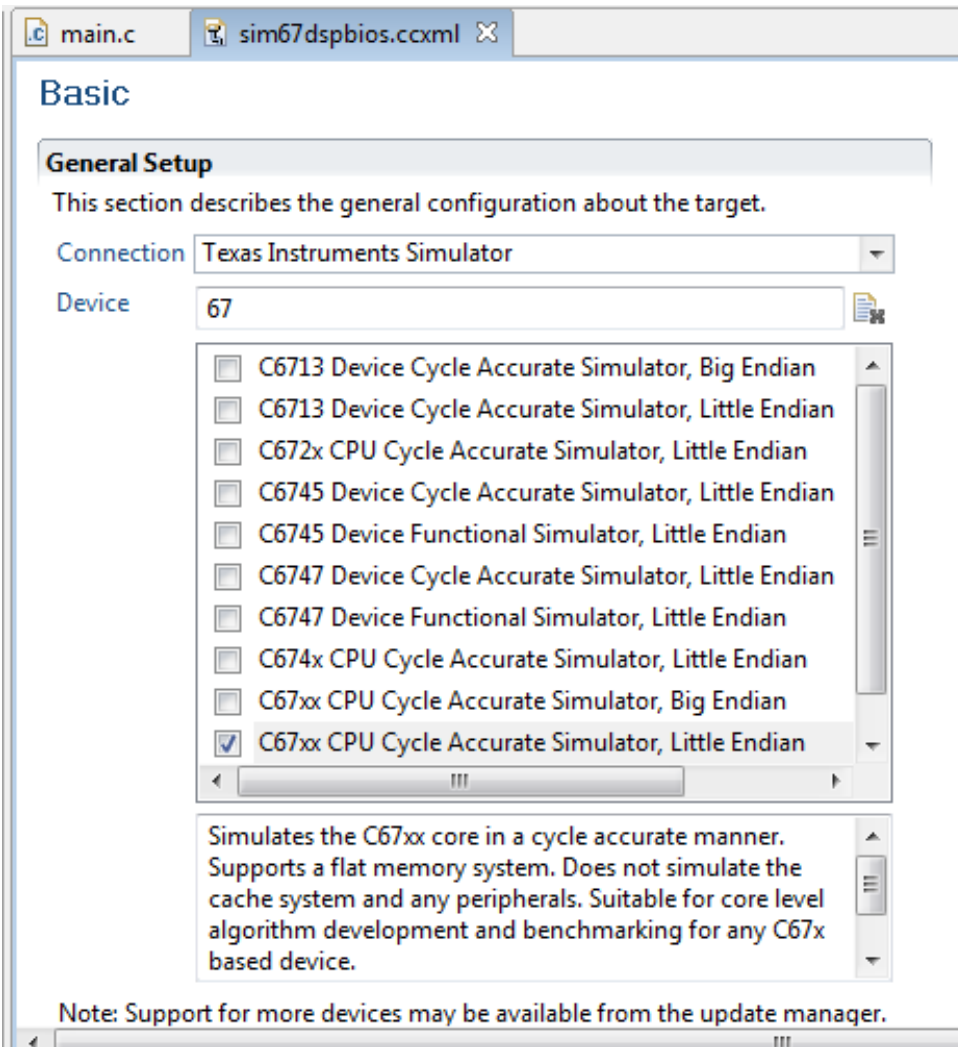
Open new target configuration file

In the New Target Configuration dialog enter the configuration file name ending in the **.ccxml** file extension. Put the file in the project directory. Select **Finish** and a new target configuration file opens.



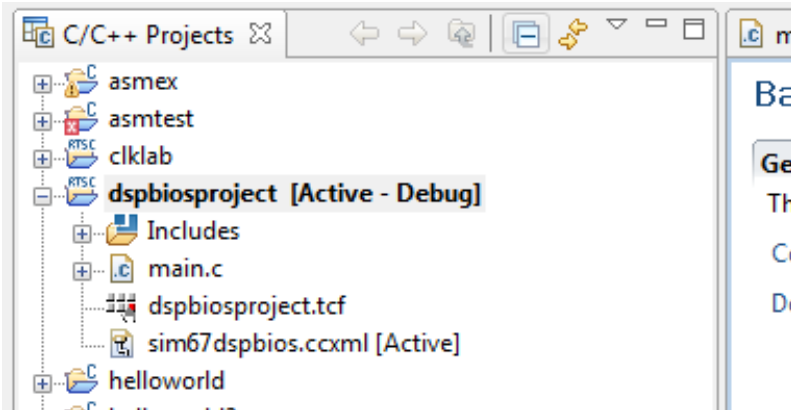
Target configuration dialog

In the target configuration, select the connection needed (simulator or some type of emulator, etc.) and the type of device. In this example the TI simulator is used with the C6713 little endian simulator device. Select **Save** when done.



## Target configuration setup

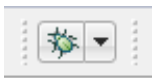
Now the target configuration is in the project.



Project files

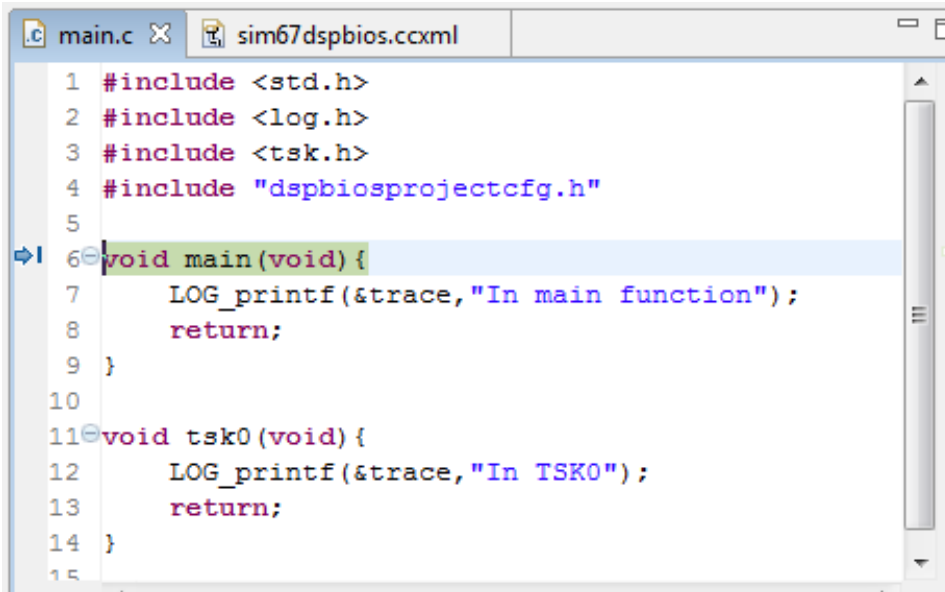
## Debug the project

To start a debug session either select **Target->Debug Active Project** or press the debug icon in the menu bar.



Debug icon

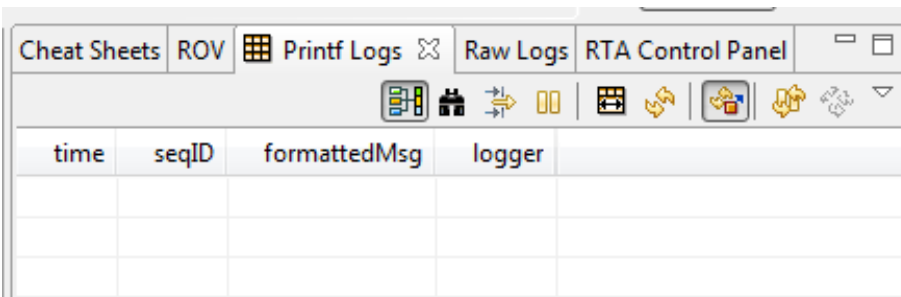
When the debug session starts the code files are compiled and linked into an executable file and the file is loaded onto the target. The initialization code is executed and the program is halted at the beginning of the **main** function.



```
1 #include <std.h>
2 #include <log.h>
3 #include <tsk.h>
4 #include "dspbiosprojectcfg.h"
5
6 void main(void) {
7     LOG_printf(&trace, "In main function");
8     return;
9 }
10
11 void tsk0(void) {
12     LOG_printf(&trace, "In TSK0");
13     return;
14 }
15
```

### Debugging session

In order to be able to see the LOG print text the real time analysis (RTA) data stream must be turned on. Select **Tools->RTA->Print Logs** which will open up the LOG print window. Select the **Stream RTA data** button on the right part of the menu bar. The icon is depressed in the figure.



### Turn on the RTA data stream

Icons on the debug window can be used to run the program and other debugging tools.



## Debugging tools

Press the green arrow to run the program. The `main` function will be executed and the program will leave the `main` function. After the `main` function executes the `TSK0` object is schedule and the `tsk0` function is executed. Both functions print to the `trace` LOG object. The output can be viewed in the **Printf Logs** window.

Cheat Sheets ROV Printf Logs Raw Logs RTA Control Panel				
time	seqID	formattedMsg	logger	
0	0	In main function	trace	
1	1	In TSK0	trace	

## Printf Logs output window

## Code Composer Studio v4 DSP/BIOS and C6713 DSK

This module describes the TI C6713 DSK hardware and how to set up a Code Composer Studio v4 DSP/BIOS project for the DSK.

### Introduction

This module describes the TMS320C6713 DSK development board and how to use it in a Code Composer Studio (CCS) v4 project that uses DSP/BIOS. An example project is included.

### Reading

- TMS320C6713 DSK Technical Reference
- SLWS106D: TLV320AIC23 Technical Manual
- SPRA677: A DSP/BIOS AIC23 Codec Device Driver for the TMS320C6713 DSK
- SPRU616: DSP/BIOS Driver Developer's Guide
- SPRA846: A DSP/BIOS EDMA McBSP Device Driver for TMS320C6x1x DSPs

### Project Files

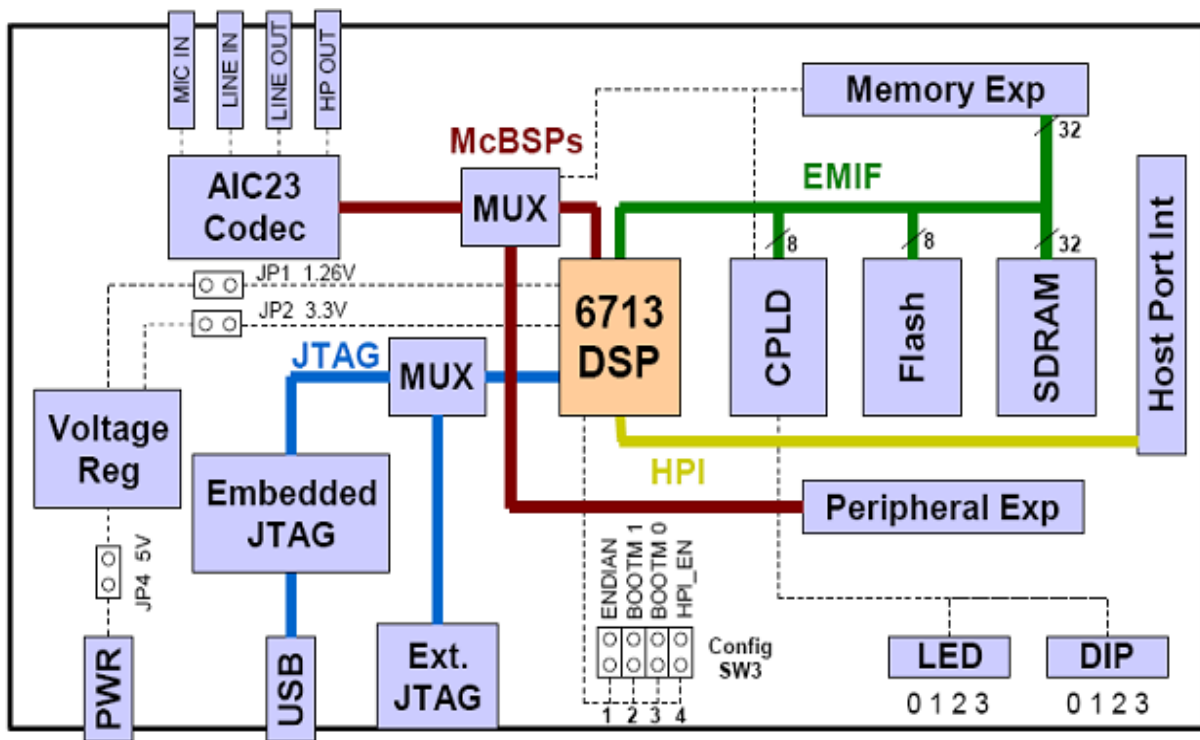
The files referred to in this module can be found in this ZIP file:

[DSK6713 audio.zip](#)

### DSK Hardware

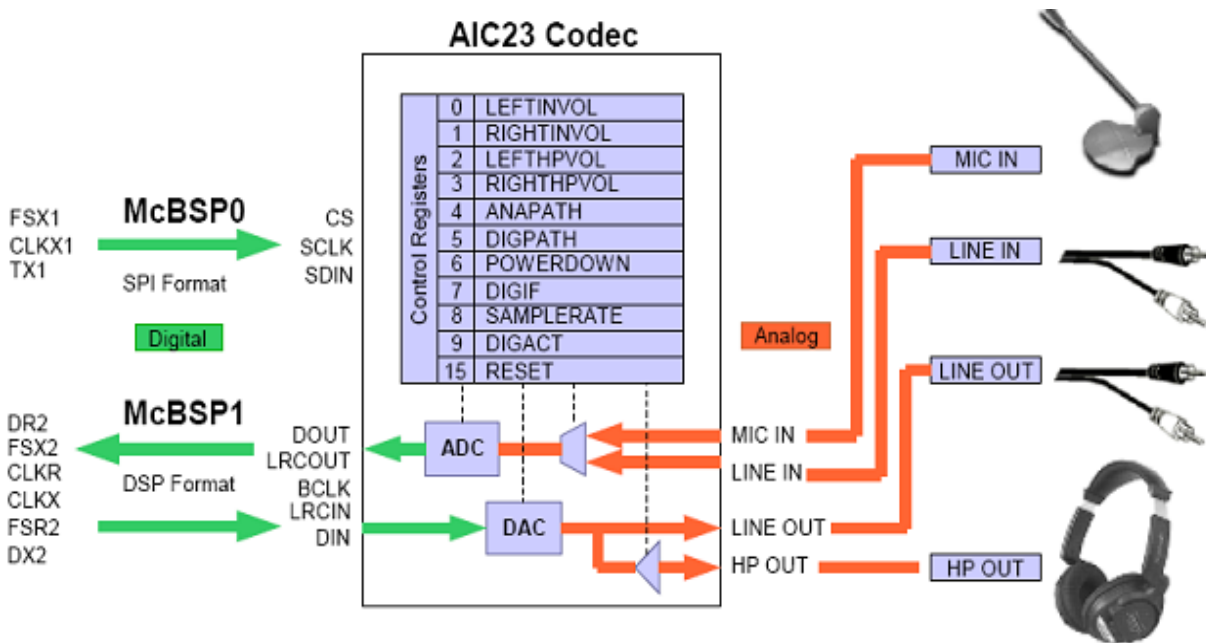
The following figure shows the block diagram of the TMS320C6713 DSK hardware. The heart of the DSK is the TMS320C6713 DSP chip which runs at 225 MHz. The DSP is in the center of the block diagram and connects to external memory through the EMIF interface. There are several devices connected to this interface. One device is a 16 Mbyte SDRAM chip. This memory, along with the internal DSP memory, will be where code and data are stored.

On the DSK board there is a TLV320AIC23 (AIC23) 16-bit stereo audio CODEC (coder/decoder). The chip has a mono microphone input, stereo line input, stereo line output and stereo headphone output. These outputs are accessible on the DSK board. The AIC23 figure shows a simplified block diagram of the AIC23 and its interfaces. The CODEC interfaces to the DSP through its McBSP serial interface. The CODEC is a 16-bit device and will be set up to deliver 16-bit signed 2's complement samples packed into a 32-bit word. Each 32-bit word will contain a sample from the left and right channel in that order. The data range is from -32768 to 32767.



TMS320C613 DSK Block Diagram taken from TMS320C6713 DSK Technical Reference



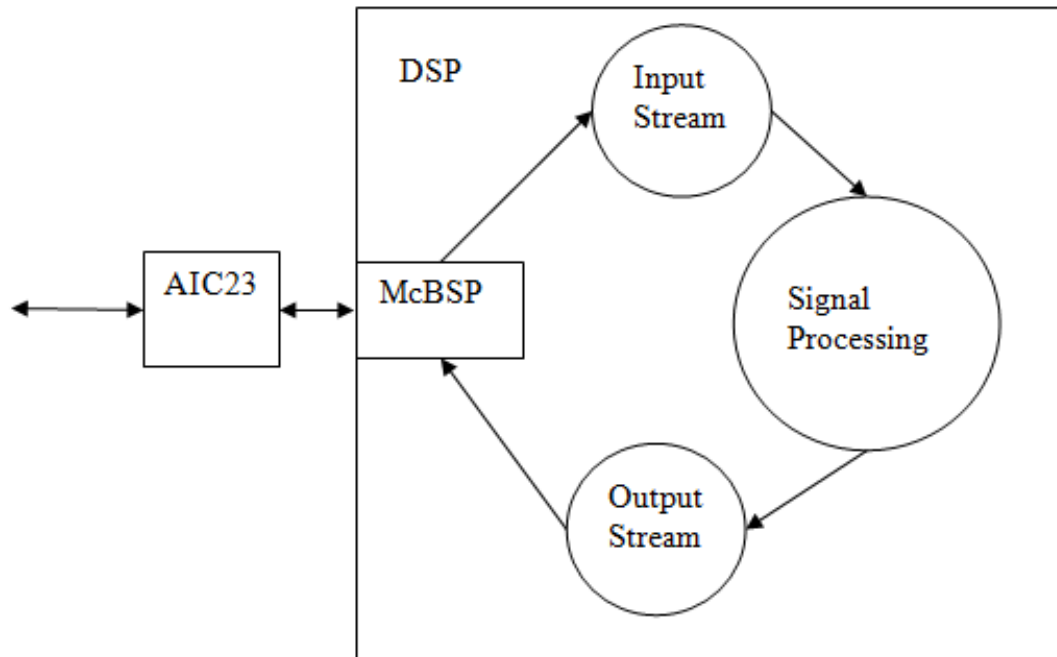


Simplified AIC23 CODEC Interface taken from TMS320C6713 DSK Technical Reference

## DSK6713 Audio Project Framework

The following figure shows a diagram of the software that will be used in this module. Texas Instruments has written some drivers for the McBSP that get data from the AIC23 and write data to the AIC23. The input data is put into an input stream (input buffer) and the output data is read from an output stream (output buffer). The signal processing software simply needs to get a buffer from the input stream, process the data, then put the resulting buffer in the output stream.

The project is set up using DSP/BIOS, a real time operating systems developed by TI. This module does not explain how to use DSP/BIOS but it will explain what objects are used in this project. The main objects are an input stream, `inStream`, an output stream, `outStream`, and a task, `TSK_processing`, which uses the function `processing()`.



Block diagram of the signal processing framework.

There are a few things that can be easily modified by the user in the main program file `DSK6713_audio.c`.

## CODEC Setup

The CODEC can be set up for different inputs and different sample rates. In the program there is a variable that sets up the registers in the CODEC:

```

DSK6713_EDMA_AIC23_DevParams AIC23CodecConfig = {
    0x0000AB01, //VersionId
    0x00000001, //cacheCalls
    0x00000008, //irqId
    AIC23_REG0_DEFAULT,
    AIC23_REG1_DEFAULT,
    AIC23_REG2_DEFAULT,

```

```

    AIC23_REG3_DEFAULT,
    AIC23_REG4_LINE, // Use the macro for Mic or
Line here
    AIC23_REG5_DEFAULT,
    AIC23_REG6_DEFAULT,
    AIC23_REG7_DEFAULT,
    AIC23_REG8_48KHZ, // Set the sample rate here
    AIC23_REG9_DEFAULT,
    0x00000001, //intrMask
    0x00000001 //edmaPriority
};

```

This is set up to use the Line In as the input with the macro **AIC23\_REG4\_LINE**. If the microphone input is needed then change this macro to **AIC23\_REG4\_MIC**. The sample rate is set to 48kHz with the macro **AIC23\_REG8\_48KHZ**. This sample rate can be changed by changing the line to one of the following macros:

```

AIC23_REG8_8KHZ
AIC23_REG8_32KHZ
AIC23_REG8_44_1KHZ
AIC23_REG8_48KHZ
AIC23_REG8_96KHZ

```

## Signal Processing Function

The main part of the software is an infinite loop inside the function **processing()**. The loop within the function is shown here.

```

while(1) {
    /* Reclaim full buffer from the input stream
    */
    if ((nmadus = SIO_reclaim(&inStream, (Ptr
    *)&inbuf, NULL)) < 0) {

```

```

        SYS_abort("Error reclaiming full buffer from
the input stream");
    }

    /* Reclaim empty buffer from the output stream
to be reused */
    if (SIO_reclaim(&outStream, (Ptr *)&outbuf,
NULL) < 0) {
        SYS_abort("Error reclaiming empty buffer from
the output stream");
    }

    // Even numbered elements are the left channel
(Silver on splitter)
    // and odd elements are the right channel
(Gold on splitter).
    /* This simply moves each channel from the
input to the output. */
    /* To perform signal processing, put code here
*/
    for (i = 0; i < CHANLEN; i++) {
        outbuf[2*i] = inbuf[2*i];// Left channel
(Silver on splitter)
        outbuf[2*i+1] = inbuf[2*i+1];// Right channel
(Gold on splitter)
    }

    /* Issue full buffer to the output stream */
    if (SIO_issue(&outStream, outbuf, nmadus,
NULL) != SYS_OK) {
        SYS_abort("Error issuing full buffer to the
output stream");
    }

    /* Issue an empty buffer to the input stream
*/
    if (SIO_issue(&inStream, inbuf,

```

```

SIO_bufsize(&inStream), NULL) != SYS_OK) {
    SYS_abort("Error issuing empty buffer to the
input stream");
}
}

```

This loop simply gets buffers to be processed with the `SIO_reclaim` commands, processes them, and puts them back with the `SIO_issue` commands. In this example the data is simply copied from the input buffer to the output buffer. Since the data comes in as packed left and right channels, the even numbered samples are the left channel and the odd numbered samples are the right channel. So the command that copies the left channel is:

```

outbuf[2*i] = inbuf[2*i]; // Left channel (Silver
on splitter)

```

Suppose you wanted to change the function so that the algorithm just amplifies the left input by a factor of 2. Then the program would simply be changed to:

```

outbuf[2*i] = 2*inbuf[2*i]; // Left channel (Silver
on splitter)

```

Notice that there is a `for` loop within the infinite loop. The loop is executed `CHANLEN` times. In the example code, `CHANLEN` is 256. So the block of data is 256 samples from the left channel and 256 samples from the right channel. When writing your programs use the variable `CHANLEN` to determine how much data to process.

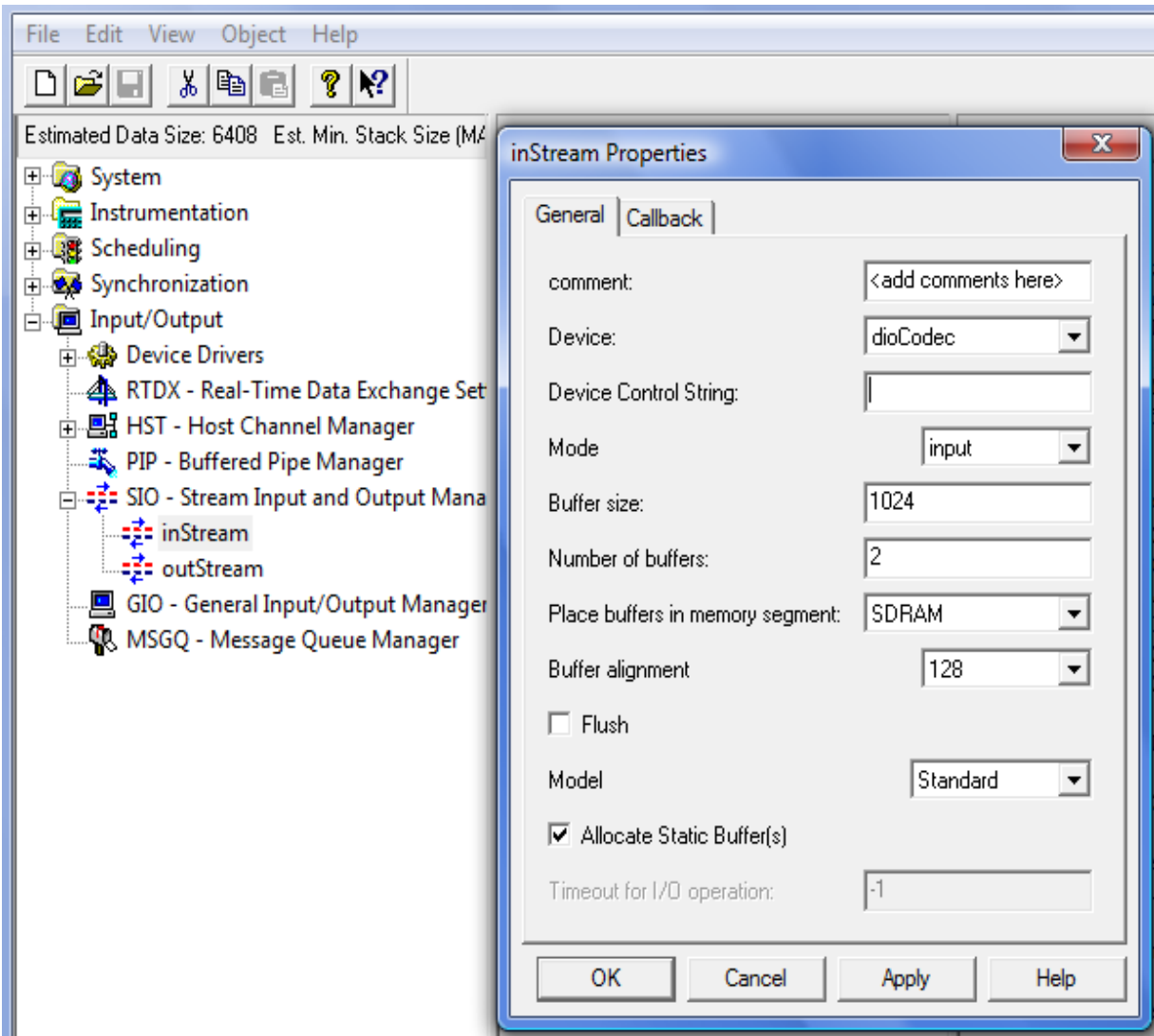
## Buffer Sizes

In the program file is a macro that is used to define the size of the buffers that are in the `inStream` and `outStream` buffers. The macro is

```
#define CHANLEN 256// Number of samples per  
channel
```

The CCS project uses TI's device drivers described in SPRA846. In the TI document SPRA846 it states that "If buffers are placed in external memory for use with this device driver they should be aligned to a 128 bytes boundary. In addition the buffers should be of a size multiple of 128 bytes as well for the cache to work optimally." If the SIO streams `inStream` and `outStream` are in external memory then the buffer sizes need to adhere to this requirement. If the streams are placed in internal memory then they can be any size desired as long as there is enough memory.

Suppose you want to keep the buffers in external memory but change the size. As an example, you want to change the length of each channel to 512 (multiple of 128). Then change `CHANLEN` to 512 and then open the configuration file `DSK6713_audio.tcf` and examine the properties for `inStream`.



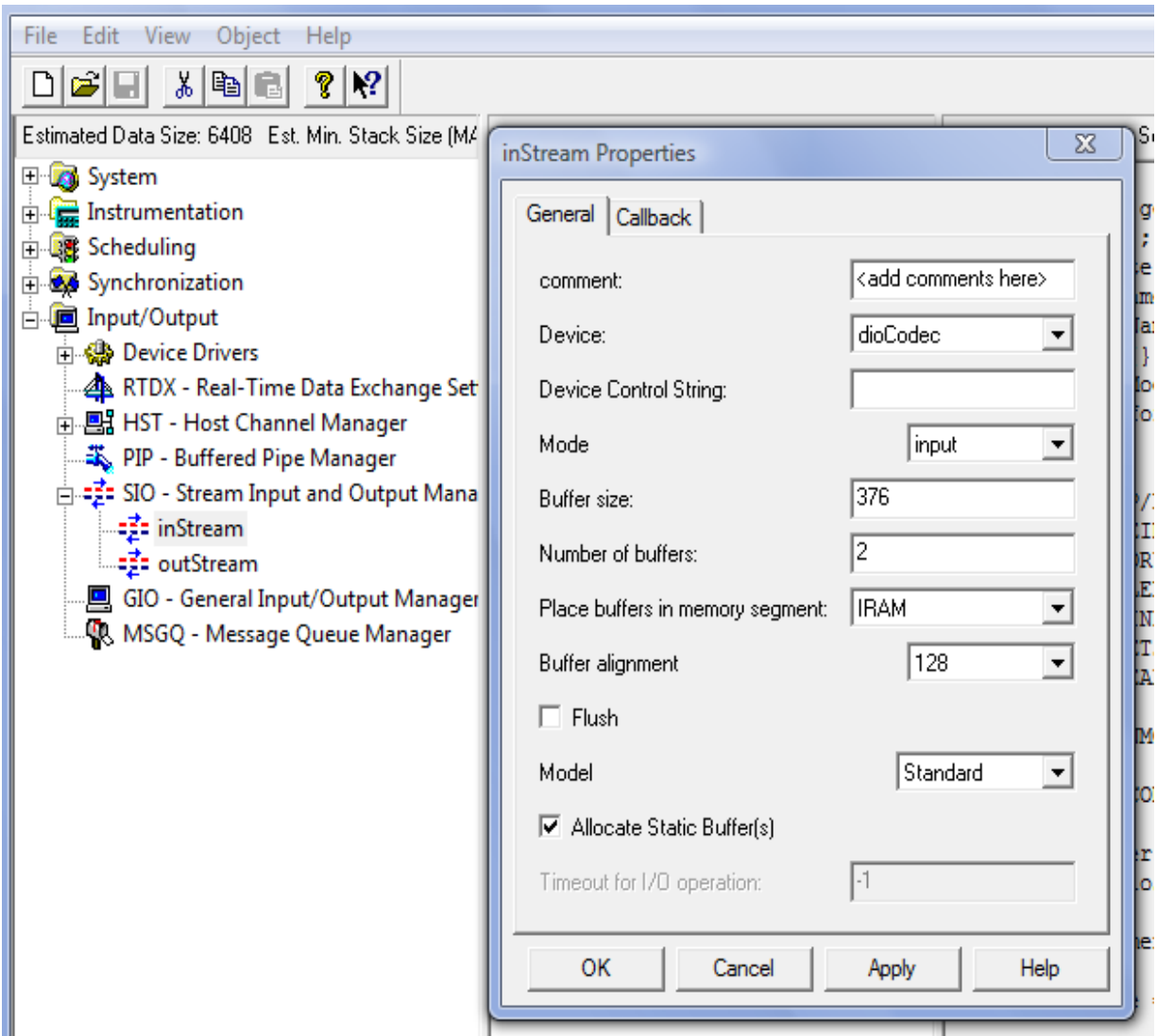
DSP/BIOS SIO object inStream properties

Notice that the buffer size is 1024. This was for a channel length of 256. In the C6713 a Minimum Addressable Data Unit (MADU) is an 8-bit byte. The buffer size is in MADUs. Since each sample from the codec is 16-bits (2 MADUs) and there is a right channel and a left channel, the channel length is multiplied by 4 to get the total buffer size needed in MADUs.  $256 * 4 = 1024$ . If you want to change the channel length to 512 then the buffer size needs to be  $512 * 4 = 2048$ . Simply change the buffer size entry to 2048. Leave the alignment on 128. The same change must be made to the

`outStream` object so that the buffers have the same size so change the buffer size in the properties for the `outStream` object also.

Suppose you have an algorithm that processes a block of data at a time and the buffer size is not a multiple of 128. To make the processing easier, you could make the stream buffer sizes the size you need for your algorithm. In this case, the stream objects must be placed in internal memory. As an example suppose you want a channel length of 94. Set the `CHANLEN` macro to 94 and then open the configuration file `DSK6713_audio.tcf` and examine and modify the properties for `inStream` and `outStream`. The buffer size will now be  $94 \times 4 = 376$  and the buffer memory segment must be change from external RAM (SDRAM) to the internal RAM (IRAM).





DSP/BIOS SIO object inStream properties showing IRAM

## Chip Support Library

The example project uses the Chip Support Library (CSL) which is an application programming interface (API) used for configuring and controlling the DSP on-chip peripherals. In order to use the CSL there are several things that must be done first.

Download and install the CSL. The CSL is labeled SPRC090 and can be found at

<http://focus.ti.com/docs/toolsw/folders/print/sprc090.html>.

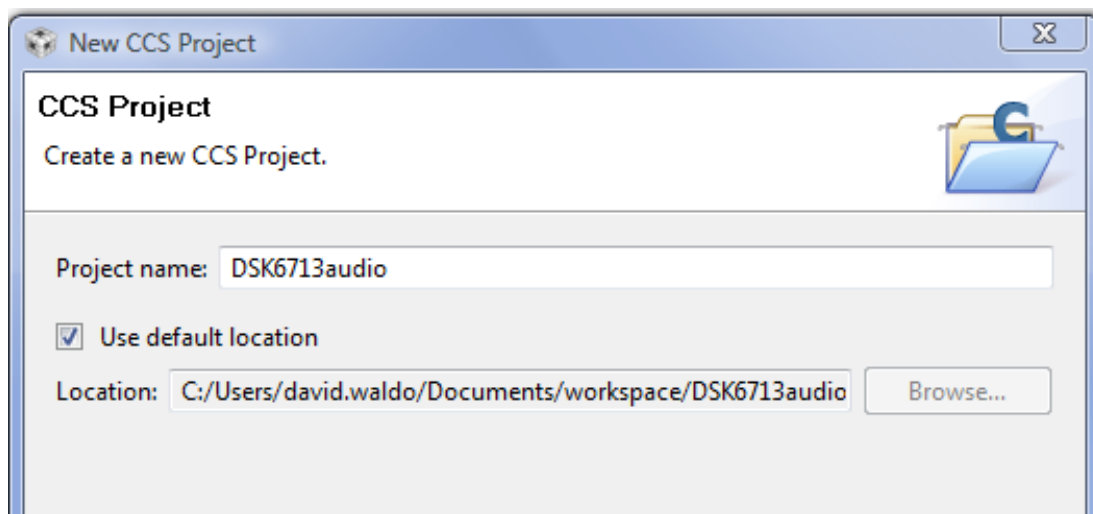
Under Windows, the CSL can be installed in the directory in your CCS install directory such as **C:\Program Files\Texas Instruments\C6xCSL** or another directory of your choosing.

## Project Setup

### Make an empty DSP/BIOS project

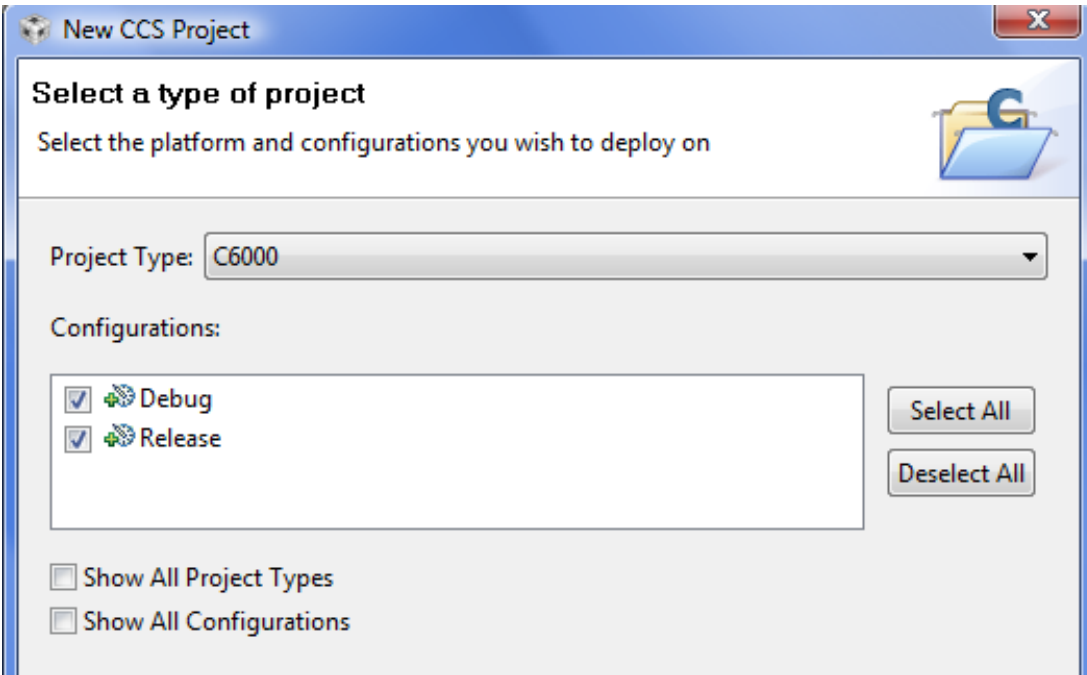
To create a CCS project select **File->New->CCS Project**.

This will bring up a window where you can enter the name of the project. The location selected is the default location for project files. Press **Next**.



CCS Project name and location

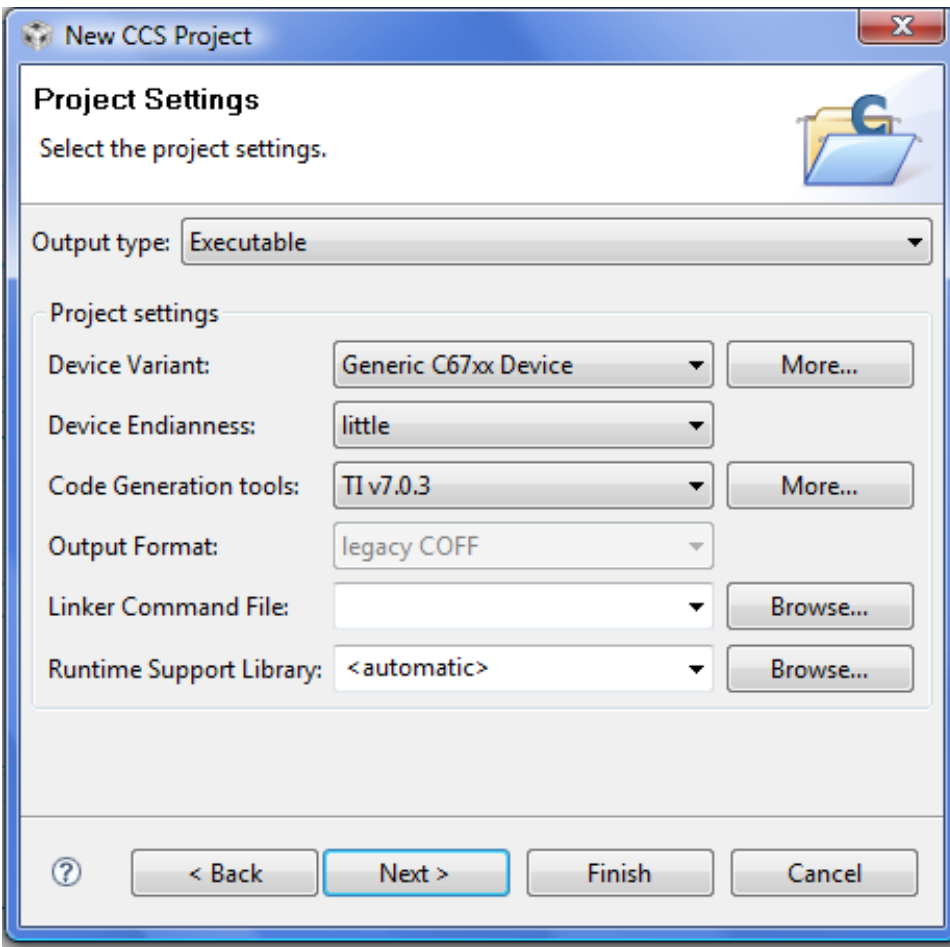
Since the example uses the TMS320C67xx processor the project type selected is **C6000**. The project configurations are **Debug** and **Release**. Select the **Next** button.



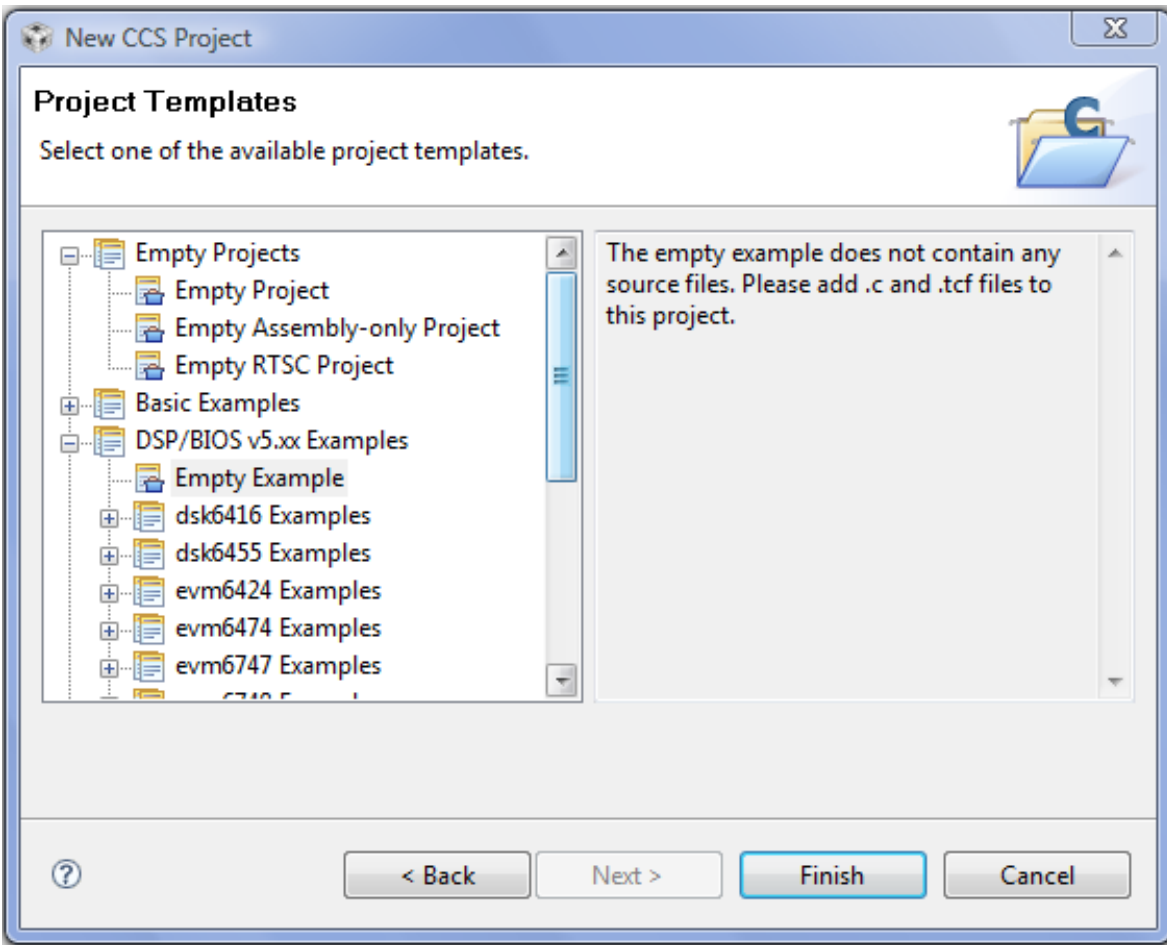
Type of CCS project

Select the **Next** button.

On the **Project Settings Screen**, select the items that pertain to the type of project to be created. Since the project will be executed select **Output Type: Executable**. The processor type is TMS320C67xx so the **Device Variant** is **Generic C67xx Device**. This project will use **Little Endian**. The code generation tools are the latest version (in this case TI v7.0.3). The runtime support library is selected to match the device variant and the endianness selected. Automatic will pick the correct library. Press **Next**.



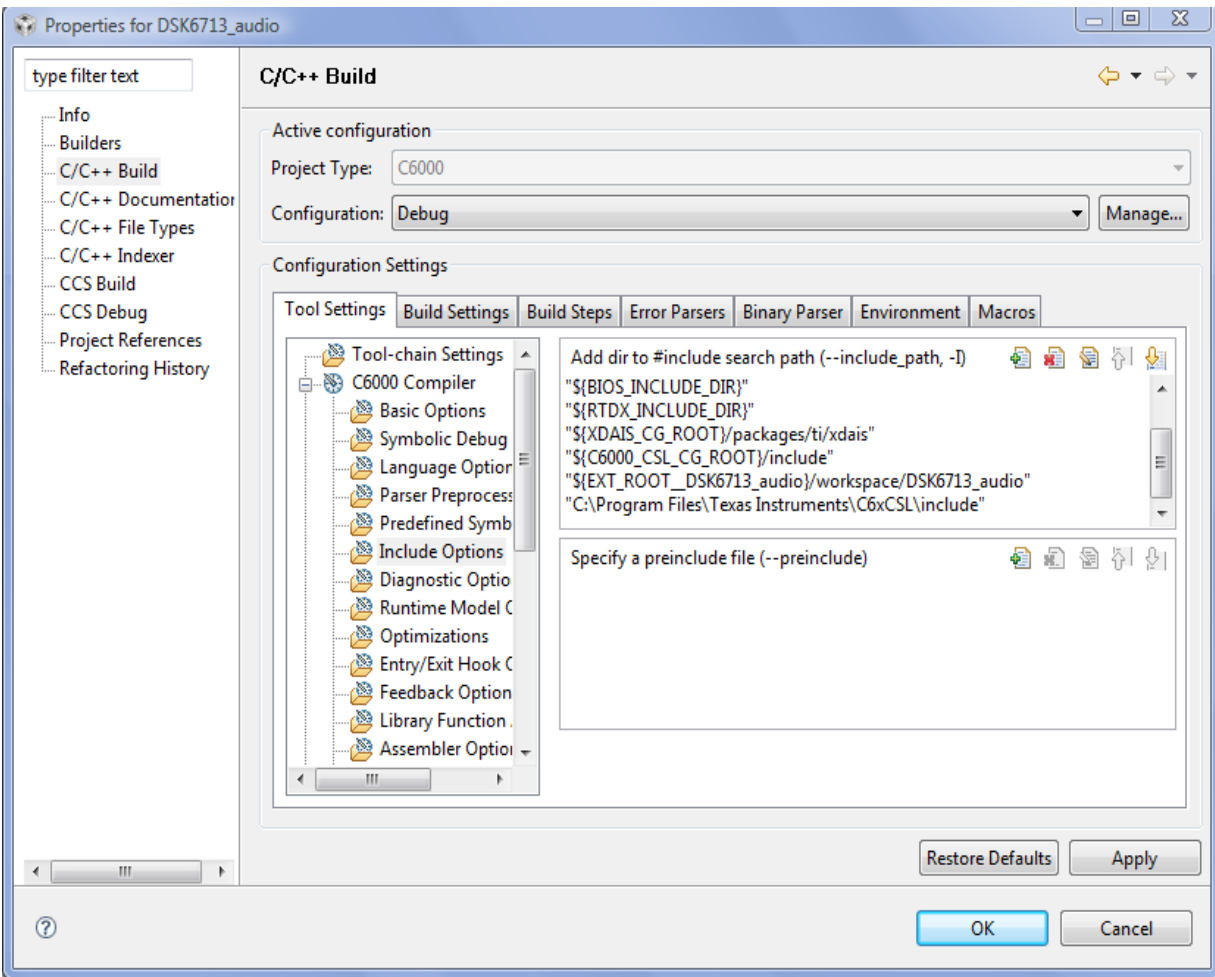
Since the project will be based on DSP/BIOS an empty DSP/BIOS example project. This will set up project parameters for DSP/BIOS.



Copy files from zip file to project directory. When they are copied to the project directory the files may be added to the project automatically but you could add them manually. There should be 5 files added to the project.

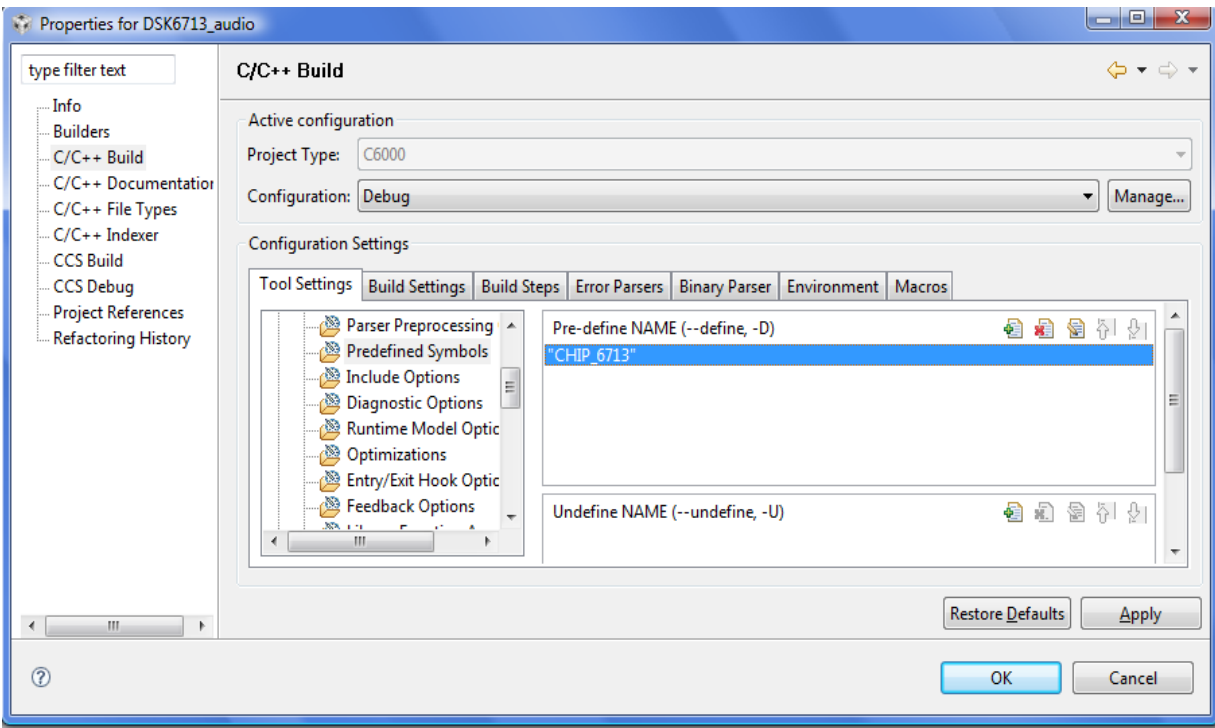
## Set up the CSL

Some options in the project need to be changed in order to use the CSL. To get the project properties select **Project->Properties**. Add the include files directory to the search path by selecting **Include Options** under **C6000 Compiler** and then clicking the green plus sign and typing in the directory name. In the example the include directory is **C:\Program Files\Texas Instruments\C6xCSL\include**.



Project properties: Include search path

The CSL needs to know what chip is being used so the chip name needs to be defined. Click on Predefined Symbols under C6600 Compiler. Click the green plus sign and add the name CHIP\_6713.

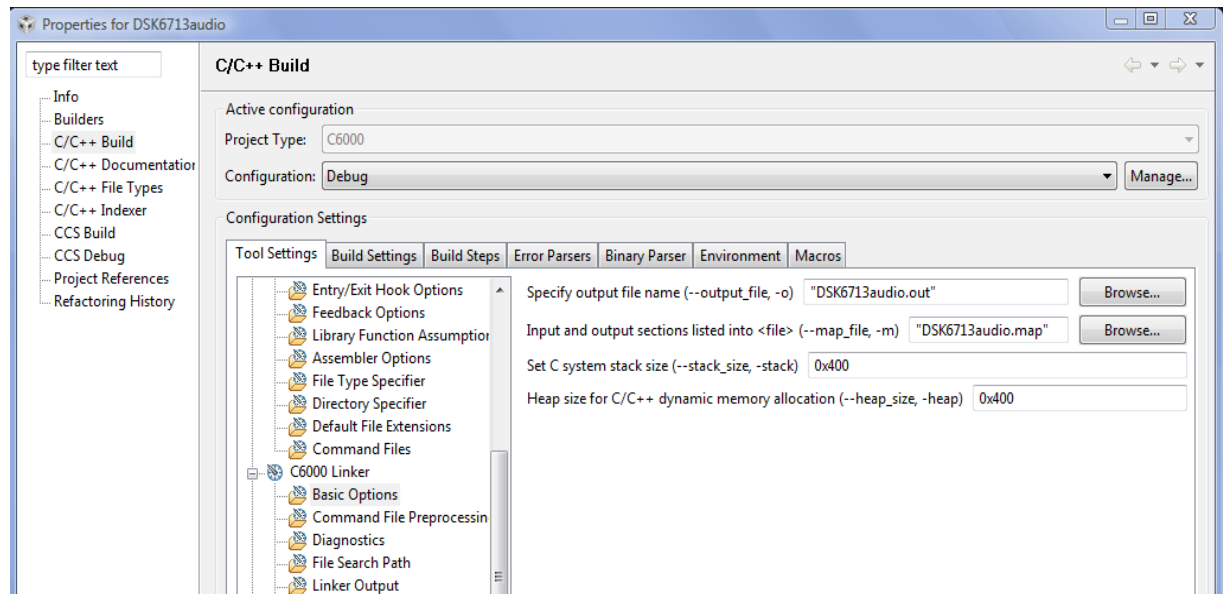


Project properties: Pre-defined name

Finally, include the chip support library file **csl6713.lib** which is in the directory **C:\Program Files\Texas Instruments\C6xCSL\lib\_3x**.

## Stack and heap sizes

To get the the project properties select Project->Properties. Change the stack and heap sizes to **0x400** by selecting **Basic Options** under **C6000 Linker**.



## Heap and stack size

### Target Configuration

In order to load the code onto the DSK board a target configuration must be created. To make a target configuration for the C6713 DSK board select **File->New->Target Configuration File** and then select the connection and board as shown in the following figure. When done, click **Save**.



## Basic

**General Setup**

This section describes the general configuration about the target.

Connection Spectrum Digital DSK-EVM-eZdsp onboard USB Emulator

Board or Device type filter text

☐ DSK5416

☐ DSK5509A

☐ DSK5510

☐ DSK6416

☐ DSK6455

☒ DSK6713

☐ DSKTCI6482

☐ Developer's Kit - Resonant DC/DC (F2808)

☐ EVM5502

☐ EVM5505

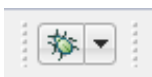
☐ EVM5515

Spectrum Digital C6713 DSK Board

## Target Configuration Setup

## Debug/test the project

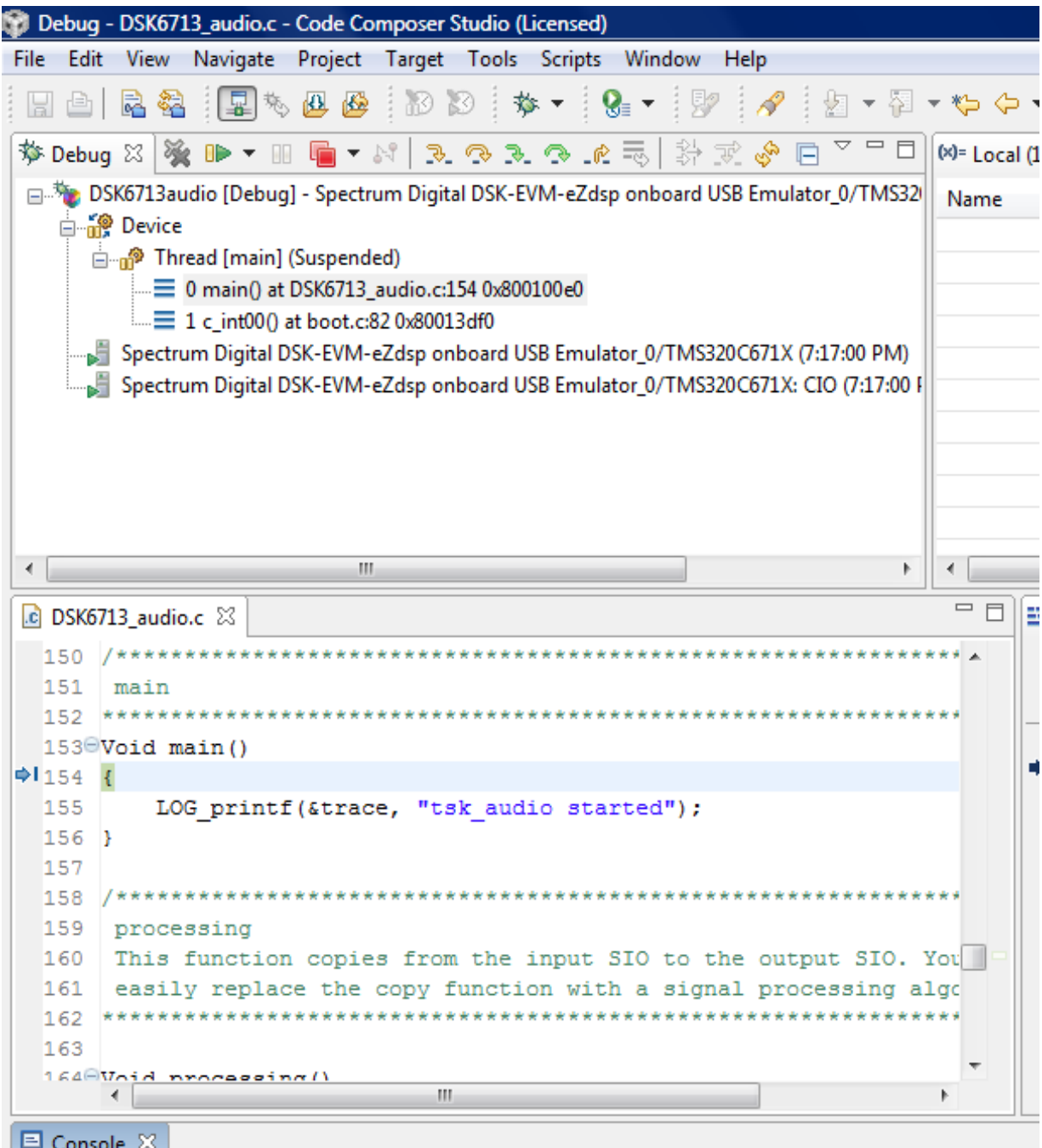
To start a debug session either select **Target->Debug Active Project** or press the debug icon in the menu bar.



Debu

g icon

When the debug session starts the code files are compiled and linked into an executable file and the file is loaded onto the target. The initialization code is executed and the program is halted at the beginning of the `main` function.



## Debug session

Press the run button and the data should be input from the selected input port and output to the LINE OUT port. Connect an appropriate input source

to the input port and connect the LINE OUT to speakers or an oscilloscope. Run the program and verify the I/O.

## Creating a TI Code Composer Studio Simulator Project

This module describes how to create a TI Code Composer Studio v3.3 project and set it up for use with the simulator. This project uses the C6713 Device Cycle Accurate Simulator but the general process can be used for other types of projects. The project is also set up to use DSP/BIOS with the configuration tool.

### Introduction

Become familiar with how to create a Code Composer Studio v3.3 project using the simulator and DSP/BIOS.

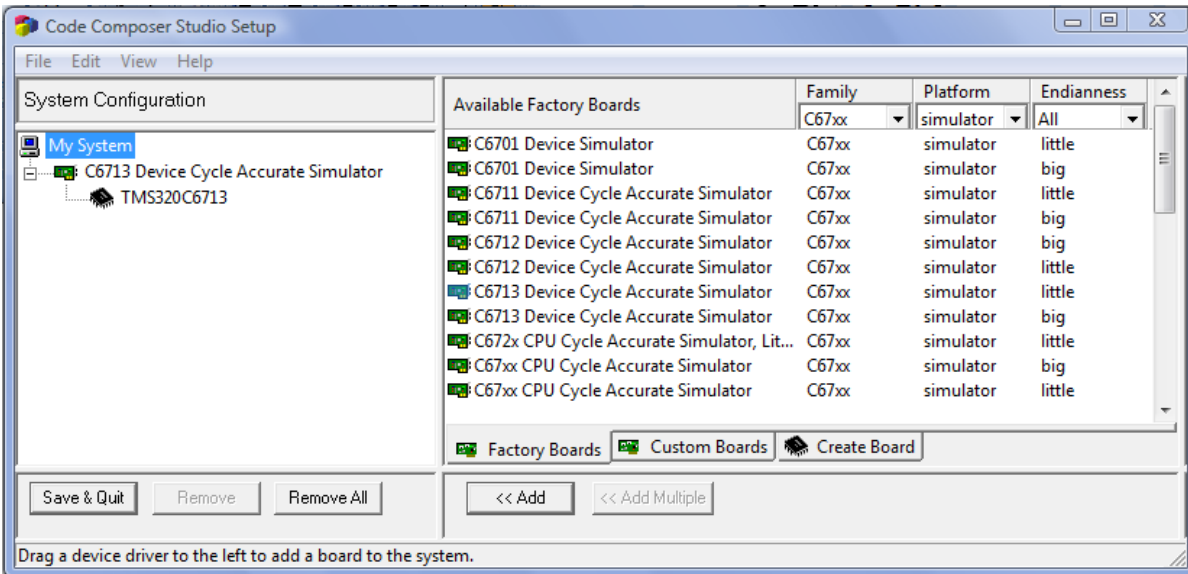
### Reading

- SPRU509: Code Composer Studio Getting Started Guide

### Description

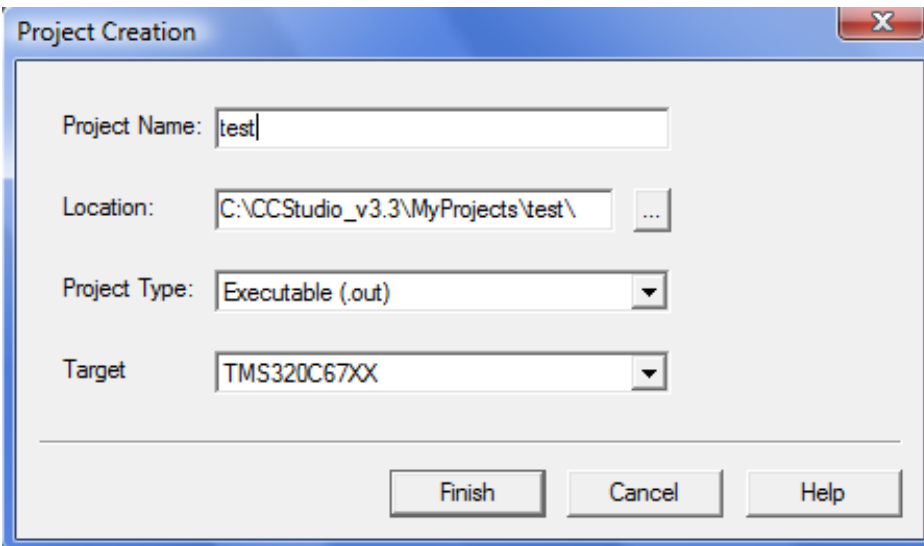
Before creating a project in Code Composer Studio (CCS), you will need to setup CCS for the board or processor you are going to be using. This module will describe how to make a project using the C6713 simulator.

- Run Setup Code Composer Studio. On the right select the C6713 Device Cycle Accurate Simulator and add it to the left. This is the configuration that will be used when CCS is started. Save and quit. Start CCS.

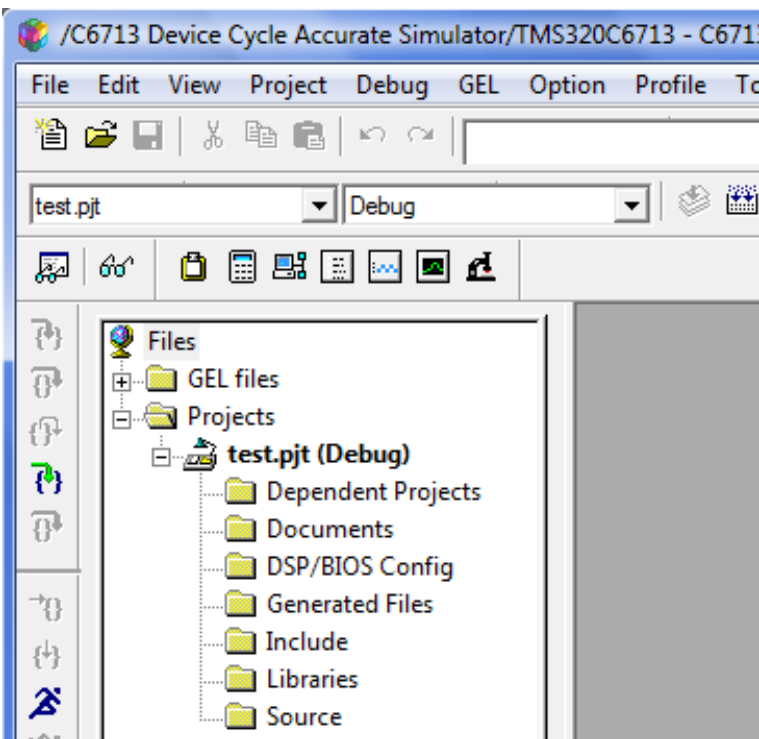


### CCS Setup for the C6713 Device Cycle Accurate Simulator.

- Create a new project in CCS by selecting Project->New... This will bring up a window that will look like Figure 2. Type the name of your project in the Project Name field and it will add a folder with the same name in the Location field. Leave everything else the same. Click Finish and CCS will open your new project. This will look like Figure 3.

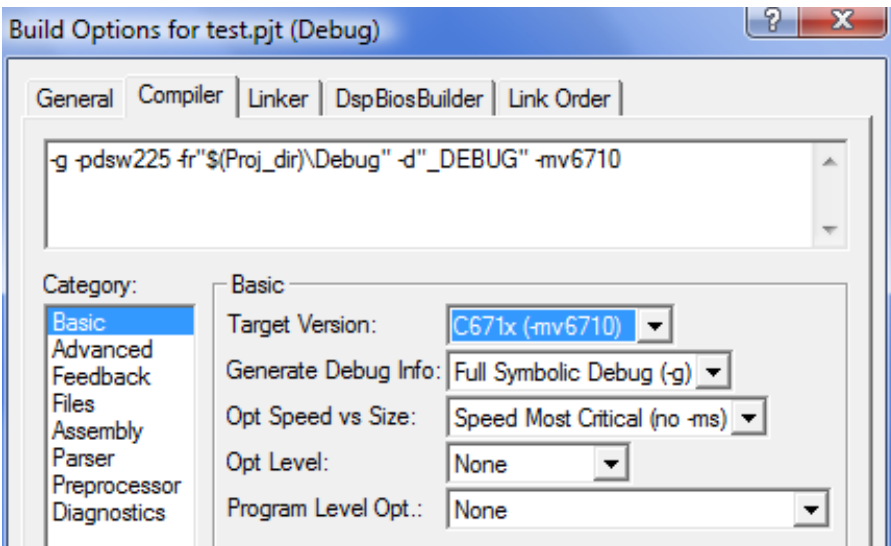


Project creation window



Project window

- After the project is created, make sure the correct processor is chosen in the build option. Select Project->Build Options... and select the processor that applies to the the configuration you set up in the Code Composer Setup tool. Here the C671x is selected to match the C6713 Device Cycle Accurate Simulator.



Processor selection in Build Options

- In order to build a project you must have files in it. To add files you must first create the files and save them. Create a new assembly code file (\*.asm), C code file (\*.c) or command file (\*.cmd) by selecting File->New->Source File. When you save the file give it the correct extension. Add the files to the project by selecting Project->Add Files To Project... and selecting the files. You can see the files in your project by expanding the "+" signs. All projects will need source files and a command file.
- For projects that use DSP/BIOS, create a new DSP/BIOS Configuration file for the hardware you are using. Select File->New-

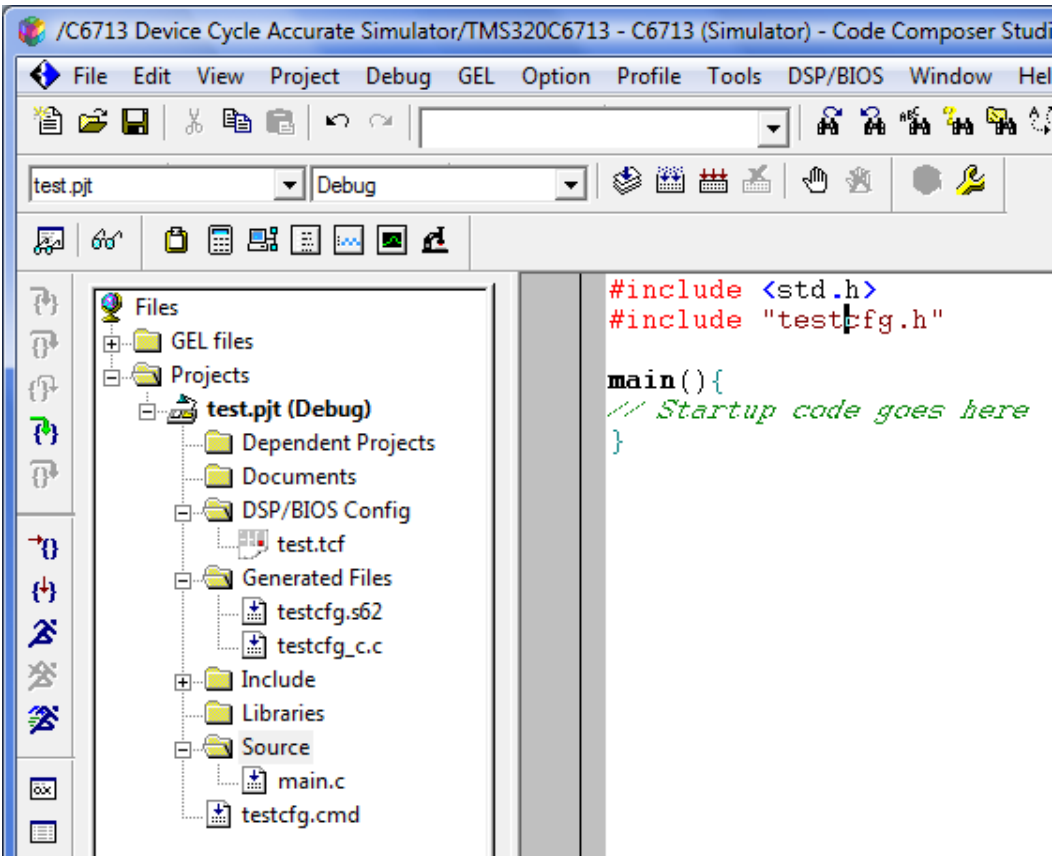


>DSP/BIOS Configuration ... If you are using the C6713 simulator then select the ti.platforms.sim67xx template.

- If you use the template there may be an error in the default settings. If there is no heap set up in any memory bank this must be done first. Click on System->MEM and check the Segment for DSP/BIOS Objects and Segment for malloc()/free(). If they say MEM\_NULL you need to set up a memory heap. Go to the SDRAM under MEM, select the properties and check Create a heap in this memory. The size should be 0x00008000. Then go back to System->MEM and change the MEM\_NULLs to SDRAM.
- Save the file as configuration file test.tcf and add it to your project. Also add the command file test.cmd file to your project.
- When developing your main C program for your project it should start with the following code example. Note that the `testcfg.h` file is created when the `test.tcf` file is compiled. Save the following code in `main.c` and add it to your project.

```
#include <std.h>
#include "testcfg.h"

main(){
// Startup code goes here
}
```



Project with tcf and cmd files added

- Set up the project for building your assembly programs by selecting Project->Build Options, clicking on the Linker tab and selecting No Autoinitialization for the Autoinit Model. When you build your C programs you will want to select Load-time Initialization.
- Once you have all your files in your project you can build the project to produce the object file (\*.out). Select Project->Build to incrementally build the project or Project->Build All to rebuild all project files. The Project->Build will only perform functions that are needed to bring the project up to date. This option will usually be quicker than the Project->Build All option.
- Now that the project has been built you must load the object file onto the target board or simulator. Do this by selecting File->Load Program

and selecting the \*.out file. The file may be located in a Debug subdirectory.

- With the project loaded you can step through the code (Debug->Step Into), view the registers (View->CPU Registers->Core Registers) and debug the results. There are buttons on toolbars that you should become familiar with to aid in the debugging process.
- Since you will be often times loading a program immediately after building it you can set up an option so that the load will occur after you build your project. Select Option->Customize... and click on the Program/Project/CIO tab. Make sure the Load Program After Build box is checked.

## Code Composer Studio v3.3 DSP/BIOS and C6713 DSK

This module describes the TI C6713 DSK hardware and how to set up a Code Composer Studio v3.3 DSP/BIOS project for the DSK.

### Introduction

This module describes the TMS320C6713 DSK development board and how to use it in a Code Composer Studio (CCS) v3.3 project that uses DSP/BIOS. An example project is included.

### Reading

- TMS320C6713 DSK Technical Reference
- SLWS106D: TLV320AIC23 Technical Manual
- SPRA677: A DSP/BIOS AIC23 Codec Device Driver for the TMS320C6713 DSK
- SPRU616: DSP/BIOS Driver Developer's Guide
- SPRA846: A DSP/BIOS EDMA McBSP Device Driver for TMS320C6x1x DSPs

### Project Files

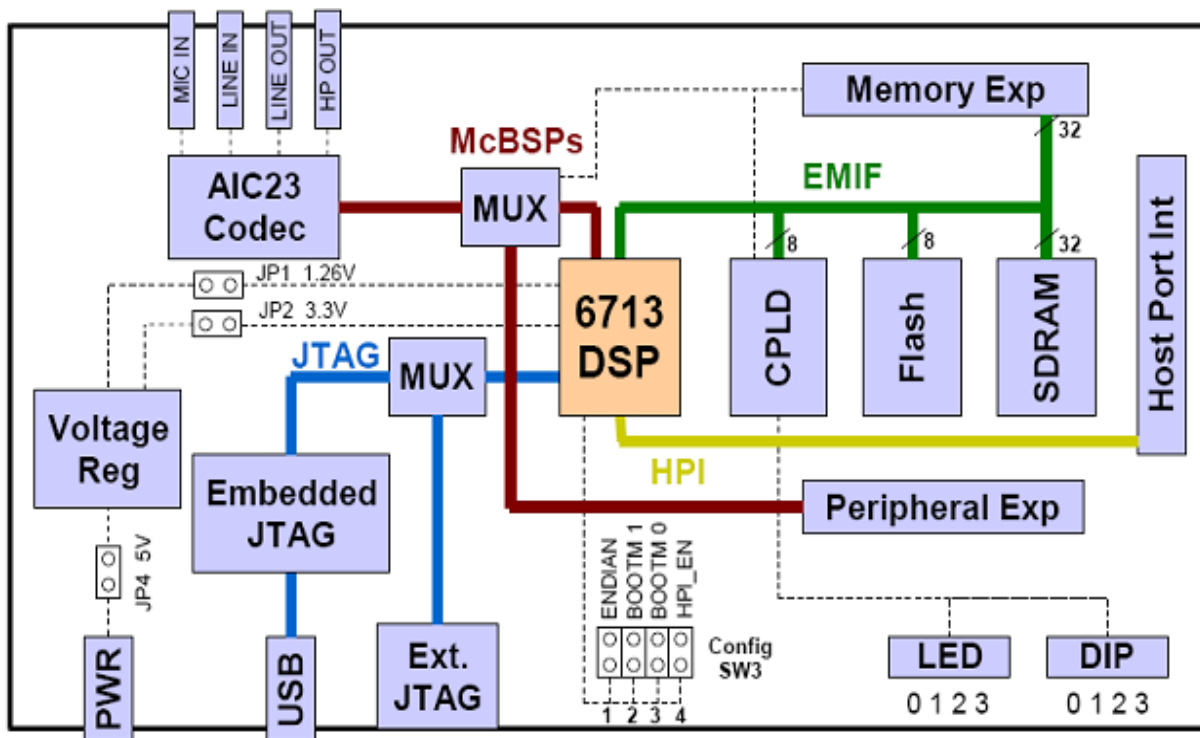
The files referred to in this module can be found in this ZIP file:

[DSK6713 audio.zip](#)

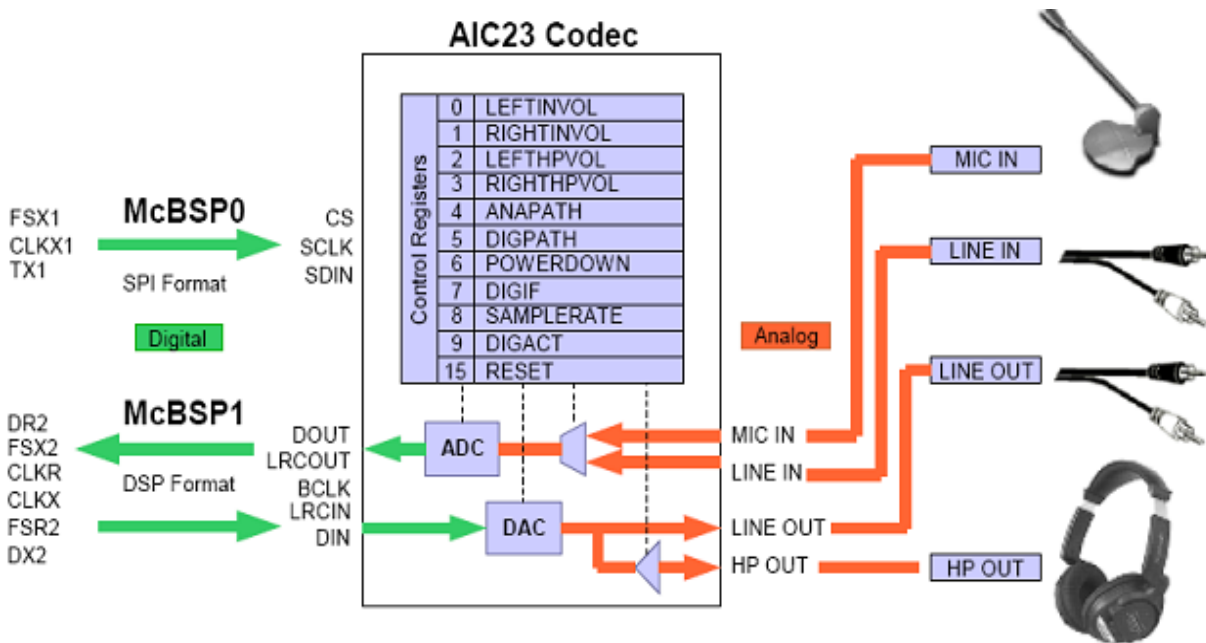
### DSK Hardware

The following figure shows the block diagram of the TMS320C6713 DSK hardware. The heart of the DSK is the TMS320C6713 DSP chip which runs at 225 MHz. The DSP is in the center of the block diagram and connects to external memory through the EMIF interface. There are several devices connected to this interface. One device is a 16 Mbyte SDRAM chip. This memory, along with the internal DSP memory, will be where code and data are stored.

On the DSK board there is a TLV320AIC23 (AIC23) 16-bit stereo audio CODEC (coder/decoder). The chip has a mono microphone input, stereo line input, stereo line output and stereo headphone output. These outputs are accessible on the DSK board. The AIC23 figure shows a simplified block diagram of the AIC23 and its interfaces. The CODEC interfaces to the DSP through its McBSP serial interface. The CODEC is a 16-bit device and will be set up to deliver 16-bit signed 2's complement samples packed into a 32-bit word. Each 32-bit word will contain a sample from the left and right channel in that order. The data range is from - to ( -1) or -32768 to 32767.



TMS320C613 DSK Block Diagram taken from TMS320C6713 DSK Technical Reference

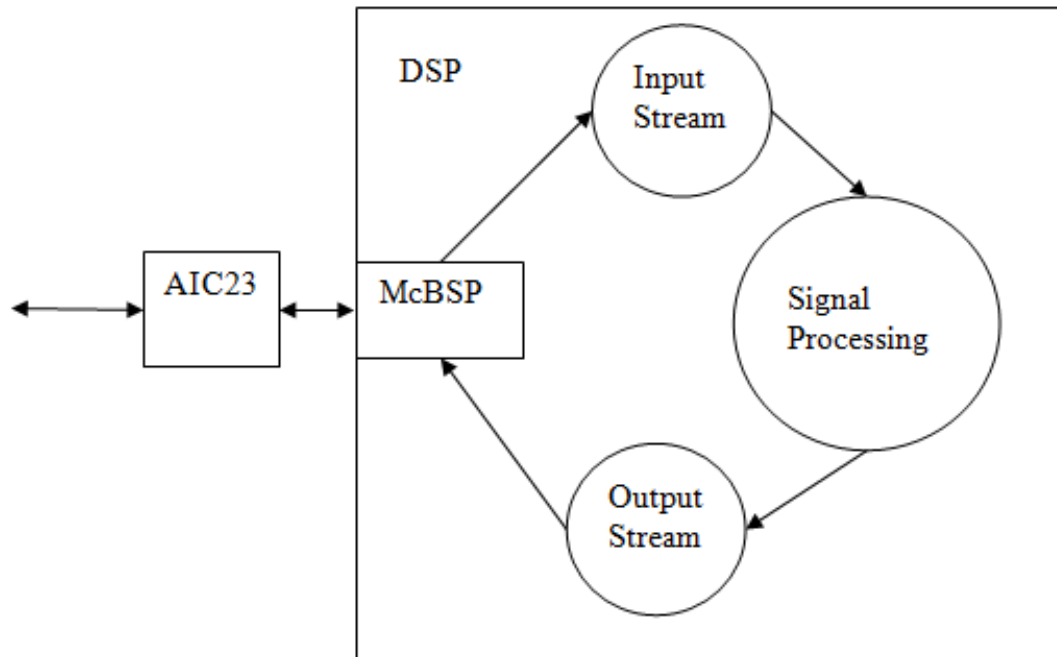


Simplified AIC23 CODEC Interface taken from TMS320C6713 DSK Technical Reference

## DSK6713 Audio Project Framework

The following figure shows a diagram of the software that will be used in this module. Texas Instruments has written some drivers for the McBSP that get data from the AIC23 and write data to the AIC23. The input data is put into an input stream (input buffer) and the output data is read from an output stream (output buffer). The signal processing software simply needs to get a buffer from the input stream, process the data, then put the resulting buffer in the output stream.

The project is set up using DSP/BIOS, a real time operating systems developed by TI. This module does not explain how to use DSP/BIOS but it will explain what objects are used in this project. The main objects are an input stream, `inStream`, an output stream, `outStream`, and a task, `TSK_processing`, which uses the function `processing()`.



Block diagram of the signal processing framework.

There are a few things that can be easily modified by the user in the main program file [DSK6713\\_audio.c](#).

## CODEC Setup

The CODEC can be set up for different inputs and different sample rates. In the program there is a variable that sets up the registers in the CODEC:

```

DSK6713_EDMA_AIC23_DevParams AIC23CodecConfig = {
    0x0000AB01, //VersionId
    0x00000001, //cacheCalls
    0x00000008, //irqId
    AIC23_REG0_DEFAULT,
    AIC23_REG1_DEFAULT,
    AIC23_REG2_DEFAULT,

```

```

    AIC23_REG3_DEFAULT,
    AIC23_REG4_LINE, // Use the macro for Mic or
Line here
    AIC23_REG5_DEFAULT,
    AIC23_REG6_DEFAULT,
    AIC23_REG7_DEFAULT,
    AIC23_REG8_48KHZ, // Set the sample rate here
    AIC23_REG9_DEFAULT,
    0x00000001, //intrMask
    0x00000001 //edmaPriority
};

```

This is set up to use the Line In as the input with the macro **AIC23\_REG4\_LINE**. If the microphone input is needed then change this macro to **AIC23\_REG4\_MIC**. The sample rate is set to 48kHz with the macro **AIC23\_REG8\_48KHZ**. This sample rate can be changed by changing the line to one of the following macros:

```

AIC23_REG8_8KHZ
AIC23_REG8_32KHZ
AIC23_REG8_44_1KHZ
AIC23_REG8_48KHZ
AIC23_REG8_96KHZ

```

## Signal Processing Function

The main part of the software is an infinite loop inside the function **processing()**. The loop within the function is shown here.

```

while(1) {
    /* Reclaim full buffer from the input stream
    */
    if ((nmadus = SIO_reclaim(&inStream, (Ptr
    *)&inbuf, NULL)) < 0) {

```



```

        SYS_abort("Error reclaiming full buffer from
the input stream");
    }

    /* Reclaim empty buffer from the output stream
to be reused */
    if (SIO_reclaim(&outStream, (Ptr *)&outbuf,
NULL) < 0) {
        SYS_abort("Error reclaiming empty buffer from
the output stream");
    }

    // Even numbered elements are the left channel
(Silver on splitter)
    // and odd elements are the right channel
(Gold on splitter).
    /* This simply moves each channel from the
input to the output. */
    /* To perform signal processing, put code here
*/
    for (i = 0; i < CHANLEN; i++) {
        outbuf[2*i] = inbuf[2*i]; // Left channel
(Silver on splitter)
        outbuf[2*i+1] = inbuf[2*i+1]; // Right channel
(Gold on splitter)
    }

    /* Issue full buffer to the output stream */
    if (SIO_issue(&outStream, outbuf, nmadus,
NULL) != SYS_OK) {
        SYS_abort("Error issuing full buffer to the
output stream");
    }

    /* Issue an empty buffer to the input stream
*/
    if (SIO_issue(&inStream, inbuf,

```

```

SIO_bufsize(&inStream), NULL) != SYS_OK) {
    SYS_abort("Error issuing empty buffer to the
input stream");
}
}

```

This loop simply gets buffers to be processed with the `SIO_reclaim` commands, processes them, and puts them back with the `SIO_issue` commands. In this example the data is simply copied from the input buffer to the output buffer. Since the data comes in as packed left and right channels, the even numbered samples are the left channel and the odd numbered samples are the right channel. So the command that copies the left channel is:

```

outbuf[2*i] = inbuf[2*i]; // Left channel (Silver
on splitter)

```

Suppose you wanted to change the function so that the algorithm just amplifies the left input by a factor of 2. Then the program would simply be changed to:

```

outbuf[2*i] = 2*inbuf[2*i]; // Left channel (Silver
on splitter)

```

Notice that there is a `for` loop within the infinite loop. The loop is executed `CHANLEN` times. In the example code, `CHANLEN` is 256. So the block of data is 256 samples from the left channel and 256 samples from the right channel. When writing your programs use the variable `CHANLEN` to determine how much data to process.

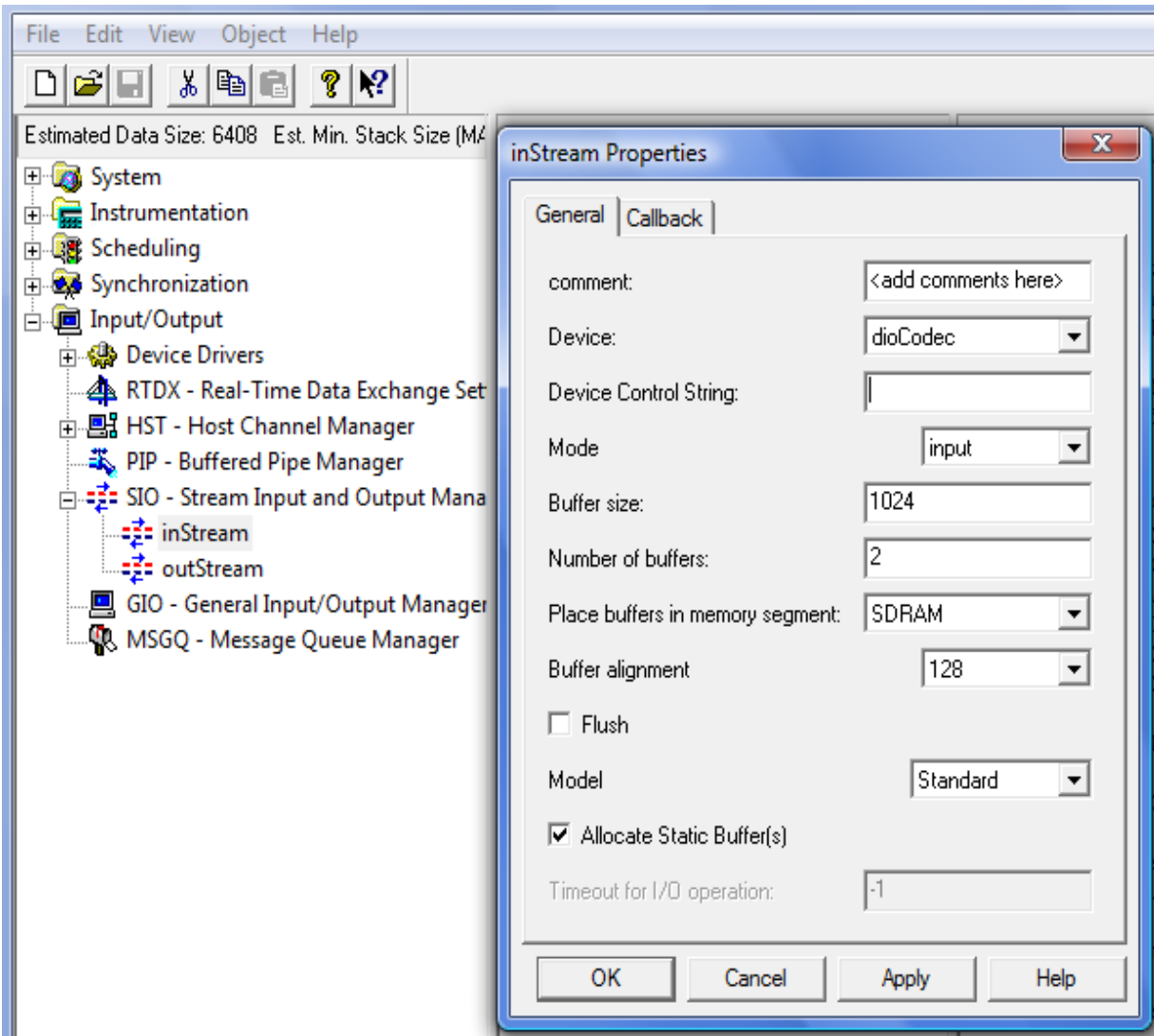
## Buffer Sizes

In the program file is a macro that is used to define the size of the buffers that are in the `inStream` and `outStream` buffers. The macro is

```
#define CHANLEN 256 // Number of samples per  
channel
```

The CCS project uses TI's device drivers described in SPRA846. In the TI document SPRA846 it states that "If buffers are placed in external memory for use with this device driver they should be aligned to a 128 bytes boundary. In addition the buffers should be of a size multiple of 128 bytes as well for the cache to work optimally." If the SIO streams `inStream` and `outStream` are in external memory then the buffer sizes need to adhere to this requirement. If the streams are placed in internal memory then they can be any size desired as long as there is enough memory.

Suppose you want to keep the buffers in external memory but change the size. As an example, you want to change the length of each channel to 512 (multiple of 128). Then change `CHANLEN` to 512 and then open the configuration file `DSK6713_audio.tcf` and examine the properties for `inStream`.

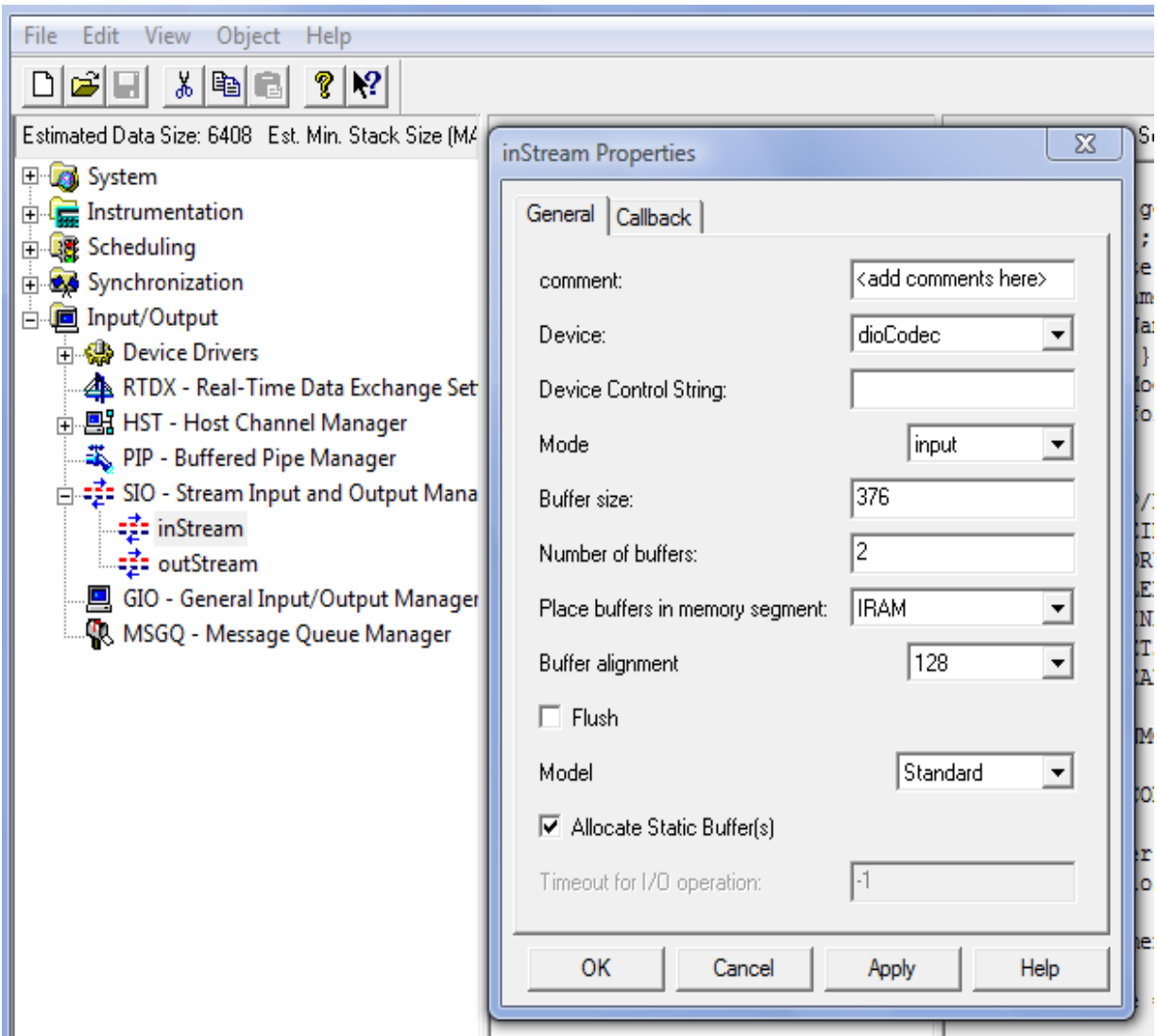


DSP/BIOS SIO object inStream properties

Notice that the buffer size is 1024. This was for a channel length of 256. In the C6713 a Minimum Addressable Data Unit (MADU) is an 8-bit byte. The buffer size is in MADUs. Since each sample from the codec is 16-bits (2 MADUs) and there is a right channel and a left channel, the channel length is multiplied by 4 to get the total buffer size needed in MADUs.  $256 * 4 = 1024$ . If you want to change the channel length to 512 then the buffer size needs to be  $512 * 4 = 2048$ . Simply change the buffer size entry to 2048. Leave the alignment on 128. The same change must be made to the

`outStream` object so that the buffers have the same size so change the buffer size in the properties for the `outStream` object also.

Suppose you have an algorithm that processes a block of data at a time and the buffer size is not a multiple of 128. To make the processing easier, you could make the stream buffer sizes the size you need for your algorithm. In this case, the stream objects must be placed in internal memory. As an example suppose you want a channel length of 94. Set the `CHANLEN` macro to 94 and then open the configuration file `DSK6713_audio.tcf` and examine and modify the properties for `inStream` and `outStream`. The buffer size will now be  $94 * 4 = 376$  and the buffer memory segment must be change from external RAM (SDRAM) to the internal RAM (IRAM).



DSP/BIOS SIO object inStream properties showing IRAM

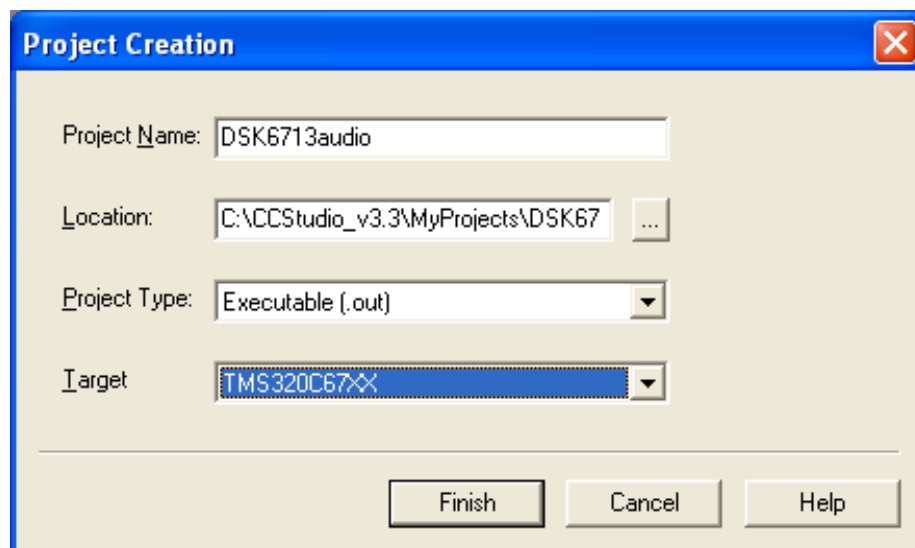
## Chip Support Library

CCS v3.3 comes with the chip support library (CSL). If for some reason you don't have it or it needs to be updated, download it from the TI website and install it.

## Project Setup

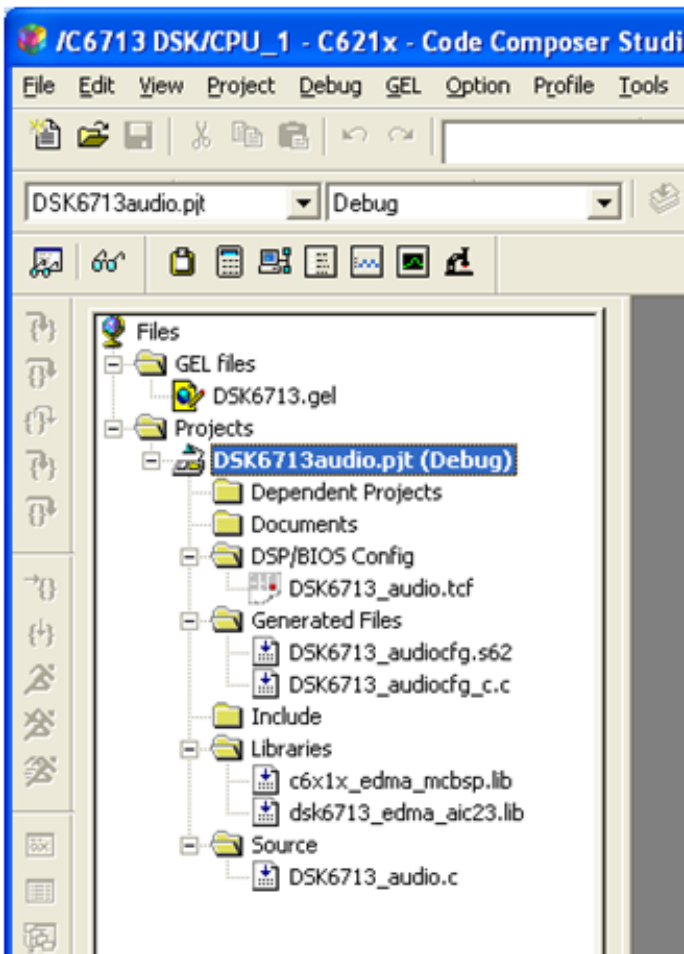
## Make an empty DSP/BIOS project

To create a CCS project select **Project->New....** This will bring up a window where you can enter the name of the project. The location selected is the default location for project files. Note that it is in the install directory for CCS v3.3. Select the Target type as **TMS320C67xx**. Press **Finish**.



CCS Project name and location

Copy files from the zip file to project directory. When they are copied to the project directory the files need to be added to the project. Right click on the project name and select **Add Files To Project....** There should be 4 files added to the project as well as a header file and a couple of automatically generated files.

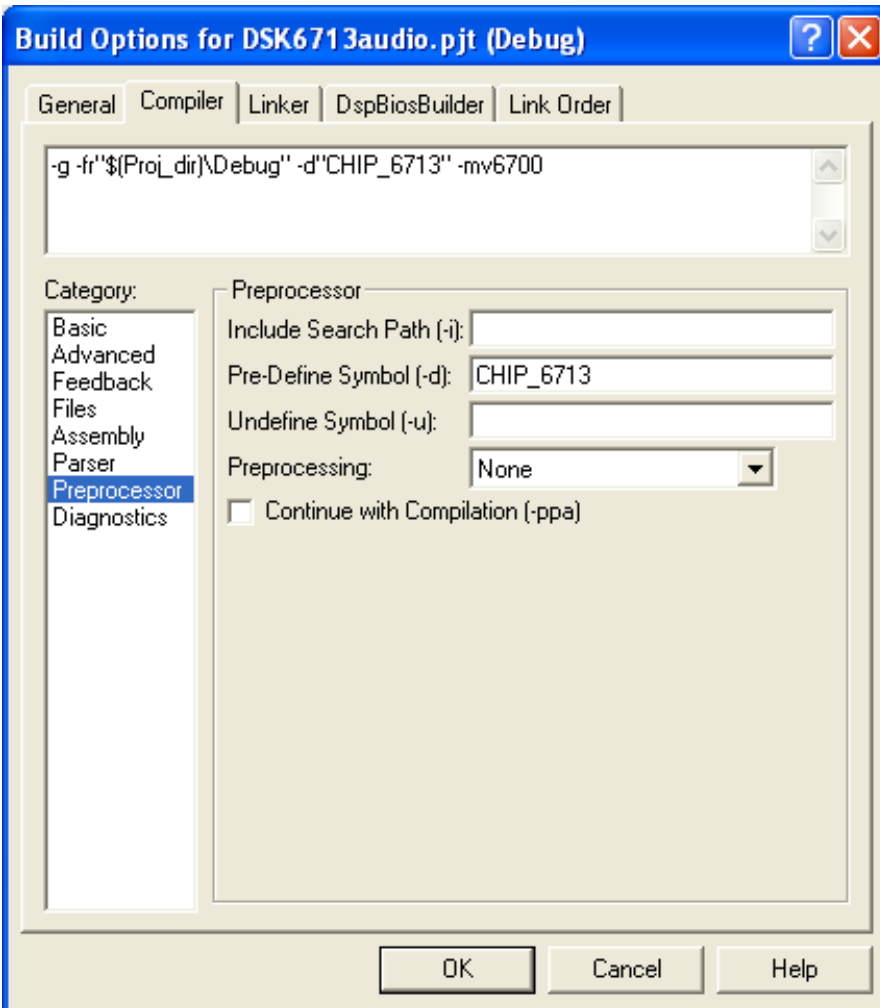


Project view after adding files

## Set up the CSL

Some options in the project need to be changed in order to use the CSL. To get to the project properties select **Project->Build Options**. Under Preprocessor enter the chip that we are using, in this case **CHIP\_6713**.





Preprocessor setting for CSL

The library file for the CSL also needs to be added to the project. Right click on the project or where it says Libraries and search for the CSL library file named **cs16713.lib**. It should be located in the directory **C:\CCStudio\_v3.3\C6000\cs1\lib**.

## Add Command File

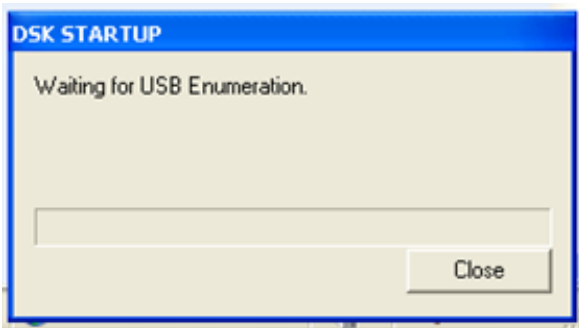
The configuration file, **DSK6713\_audio.tcf**, generates several files, one of them being a command file titled **DSK6713\_audio.cmd**. Add this file to the project. If it is not there, you may need to attempt to build the project so that the file gets generated.

## Using CCS with the DSK

### Startup CCS with DSK connected to PC

After installing CCS v3.3 the C6713 DSK drivers need to be installed also. If they are not already installed then download them from Spectrum Digital, manufacturer of the DSK, or install them from the install disk that comes with the DSK.

Add power to the DSK board and then plug the USB connector into the PC you are using. Start up CCS. As the splash screen comes, under Windows XP, there should be a little window that pops up in the bottom right corner of the screen.



Popup window when CCS starts and DSK is connected

### Connect to target

CCS v3.3 must first be connected to the target before the executable can be loaded onto the board. Select **Debug->Connect**. This will cause CCS to connect to the board and the board status should be in the lower left corner of the CCS window.



Target status

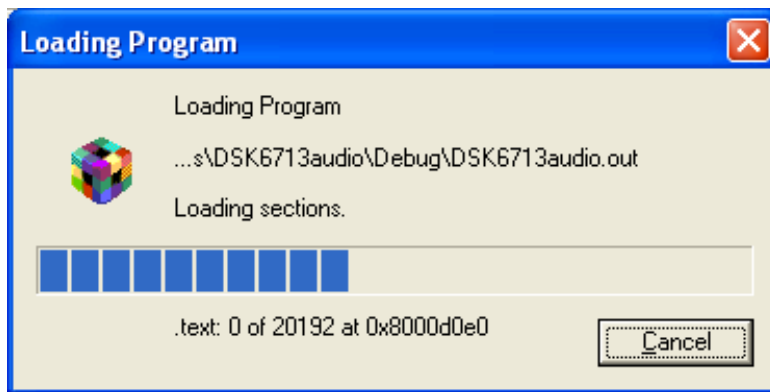
## Debug/test the project

In order to debug and run the program you need to build it and then load it onto the DSK board. To build the project, click the Incremental Build button.



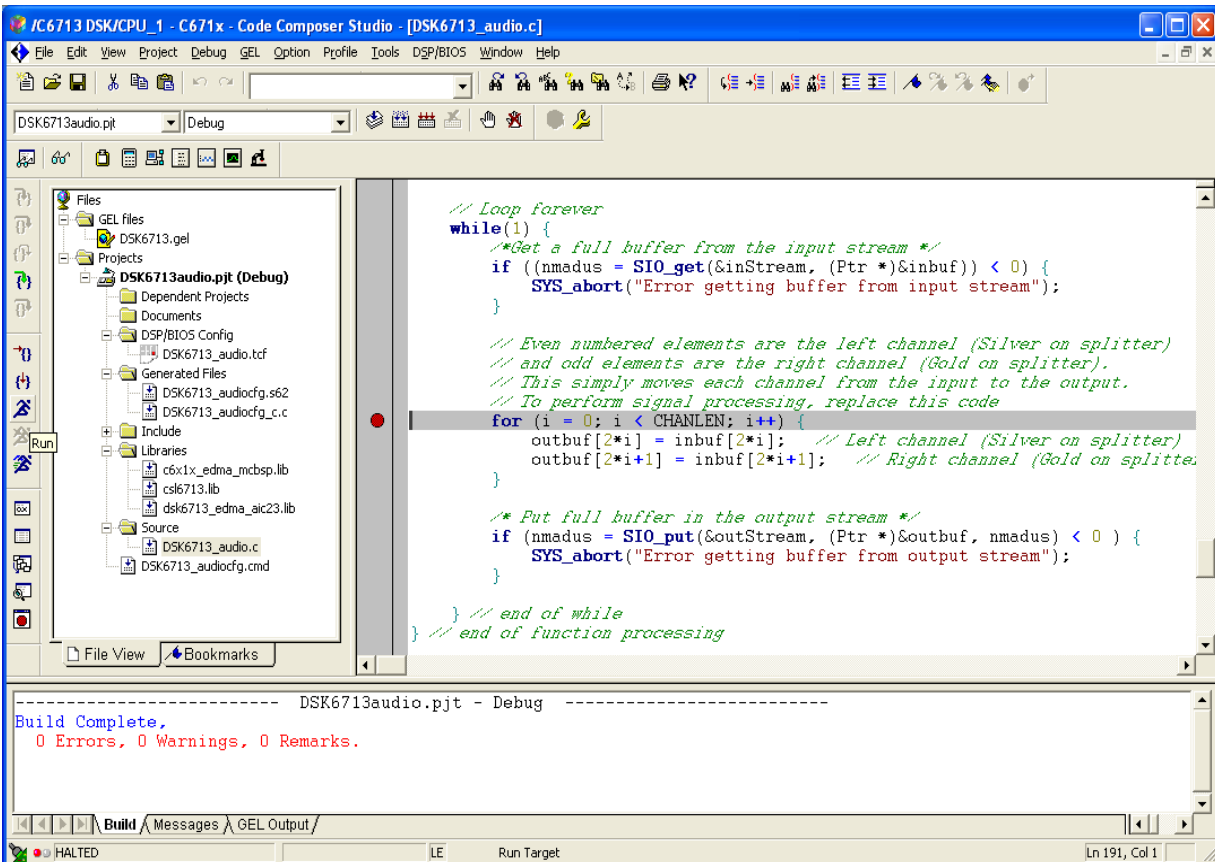
Incremental  
Build  
button

After the project builds successfully, load the **.out** file to the DSK board but selecting **File->Load Program...** and selecting the file. Or, if you want it loaded automatically, select **Options->Customize** and then on the **Project/Program/CIO** tab select the checkmark next to **Load Program After Build**. When you load the program you will see a status window like the following figure.



Load Program status window

Once the program is loaded it is ready to be used. The following figure shows the final project with the main program file.



Final project view

## TI DSP/BIOS LOG Module

This module describes the basics of the TI DSP/BIOS LOG module.

### Introduction

This module describes the basics of the TI DSP/BIOS LOG module.

### Reading

- SPRU423 TMS320 DSP/BIOS Users Guide: Read the section Event Log Manager (LOG Module)

### LOG Module

When running a real-time application it is desirable to not interrupt the processing while attempting to monitor or debug code. Printing to the standard I/O is very time consuming on the processor. The LOG module helps reduce this time. Formatting of the display takes place on the host computer rather than on the DSP. Data is stored in real-time to a buffer on the DSP and then during idle time it is transferred to the host computer. On the host computer the print statements are formatted and displayed.

The main functions used in the LOG module are `LOG_printf` and `LOG_printf4`. They work very similar to the C `printf` function. In order to print using the LOG module a LOG object must be defined. To create a new LOG object open the configuration file and right click on **Instrumentation->LOG** and select **Insert LOG**. This will create a LOG object called `LOG0`. The name of the object can be changed. On the properties window for the object the buffer length is in words and sets the length of the buffer. The buffer can be either a circular buffer, which contains the last messages written to the object, or fixed, which contains the first messages written to the buffer. A circular buffer allows messages to be sent continually. Each message writes 4 words of data. The format of the function call is

```
LOG_printf(LOG_Obj *log,String format,arg0,arg1)
LOG_printf4(LOG_Obj *log,String
format,arg0,arg1,arg2,arg3);
```

where the **String format** is a usual C **printf** format string with the conversion characters given in Table 1.

Conversion Character	Description
%d	Signed integer
%x	Unsigned hexadecimal integer
%u	Unsigned integer
%o	Unsigned octal integer
%s	Character string. This character can only be used with constant string pointers.
%r	Symbol from symbol table
%p	pointer

When using objects defined with the configuration tool the code must include definitions of the objects and the header file **log.h**. The configuration tool generates a header file that contains the definitions. For instance, the file **log.tcf** would generate the file **logcfg.h**. If an object called **LOG0** was created, then the file would contain

```
extern far LOG_Obj LOG0;
```

Include the configuration header file in your code and you won't need to make the definitions yourself. A call the the print function might look like this

```
LOG_printf(&LOG0, "Hello world");
```

To view the message log select **Tools->RTA->Printf Logs**.



## LOG Lab

The module is a lab assignment to better understand the TI DSP/BIOS LOG module.

## Introduction

This lab module will help you become familiar with TI DSP/BIOS LOG module and its use.

## Reading

- SPRU423 TMS320 DSP/BIOS Users Guide: Read the section Event Log Manager (LOG Module)

## Lab Module Prerequisites

None

## Laboratory

- Follow the procedure in [Code Composer Studio v4 DSP/BIOS Project](#) to create a new CCS DSP/BIOS project. Name the project **loglab**. Your project should have a DSP/BIOS configuration file named **loglab.tcf**.
- Create a LOG object called **trace**. Use the default length of 64 and the logtype circular.
- Add a task to the configuration. This is part of the TSK module but you don't need to know about the module to be able to do this lab. Add a task by right clicking on **Scheduling->TSK** and selecting **Insert TSK**. A new task should be added called **TSK0**.
- Specify the function the task will run by right clicking on **TSK0** and bringing up its properties. On the Function tab next to Task Function type in the function name **\_logTSK**. This will specify the C function **logTSK** as the function it will call.

- Create a `main.c` file and include a main function that does nothing and a function `logTSK`. The main structure of the file is

```
#include <std.h>
#include <log.h>
#include <tsk.h>
#include "loglabcfg.h"

main()
{
}

logTSK()
{
/* your code here */
}
```

- Write a loop in the `logTSK` function that loops 20 times and writes some text indicating that the loop number is being printed and print the loop number. For instance, have it print

```
Loop Number 0
Loop Number 1
Loop Number 2
...
```

- Start the debug session.
- Open the log view by selecting **Tools->RTA->Printf Logs**.
- Run the program and record the results.

## TI DSP/BIOS TSK Module

This module describes the basics of the TI DSP/BIOS Task Module (TSK).

### Introduction

DSP/BIOS provides several different managers manage threads of execution. One of them is the Task Manager (TSK). This module explains the basics of TSK objects and use. This module only explains the use of static TSK objects which are set up using the configuration tool.

### Reading

- SPRU423 TMS320 DSP/BIOS Users Guide: Read the section on Tasks

### TSK Module

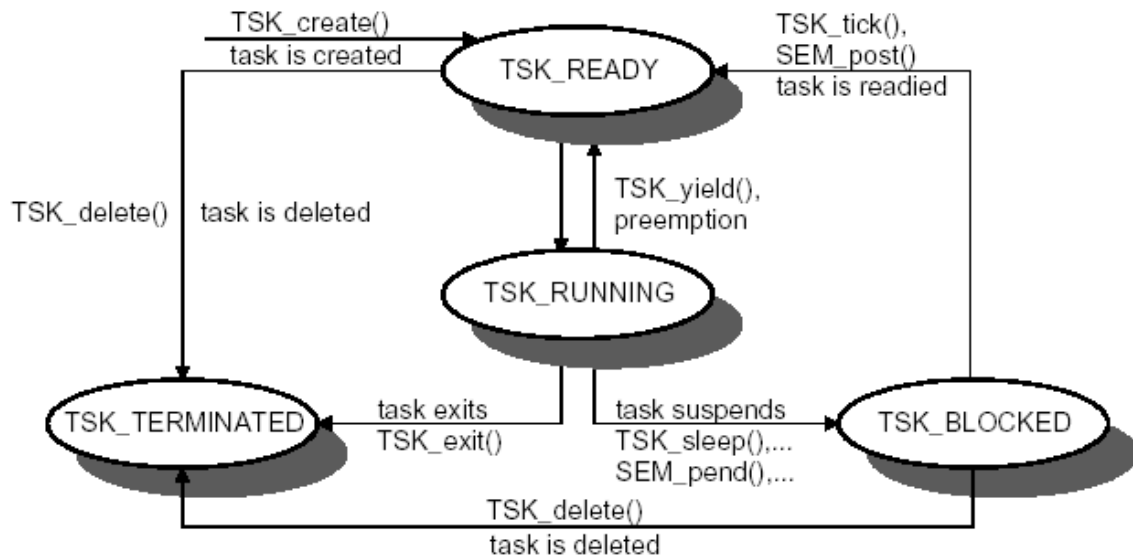
Tasks are independent threads of code that conceptually run concurrently. The processor time is shared among the tasks. Each task has a priority which is used to determine which task gets processor time.

Each task has its own stack to store local variables, nesting function calls and for saving the task state when it is preempted. The stack size can be set individually for each task.

A task is always in one of four states

- **Running**, which means the task is the one actually executing on the system's processor;
- **Ready**, which means the task is scheduled for execution subject to processor availability;
- **Blocked**, which means the task cannot execute until a particular event occurs within the system; or
- **Terminated**, which means the task is “terminated” and does not execute again.

Figure 1 shows a state transition diagram for a task. There is only one thread in the TSK\_RUNNING state. When a task is in the TSK\_RUNNING state all the other tasks in the TSK\_READY state are at the same or lower priority. When a task of higher priority enters the TSK\_READY state a preemption immediately occurs and the running task enters the TSK\_READY state and the higher priority task enters the TSK\_RUNNING state. Tasks can become blocked when a resource is unavailable or when some other event that causes blocking occurs. Semaphores are used to synchronize access to resources and can cause a task to block. When the resource becomes available the task enters the TSK\_READY state.

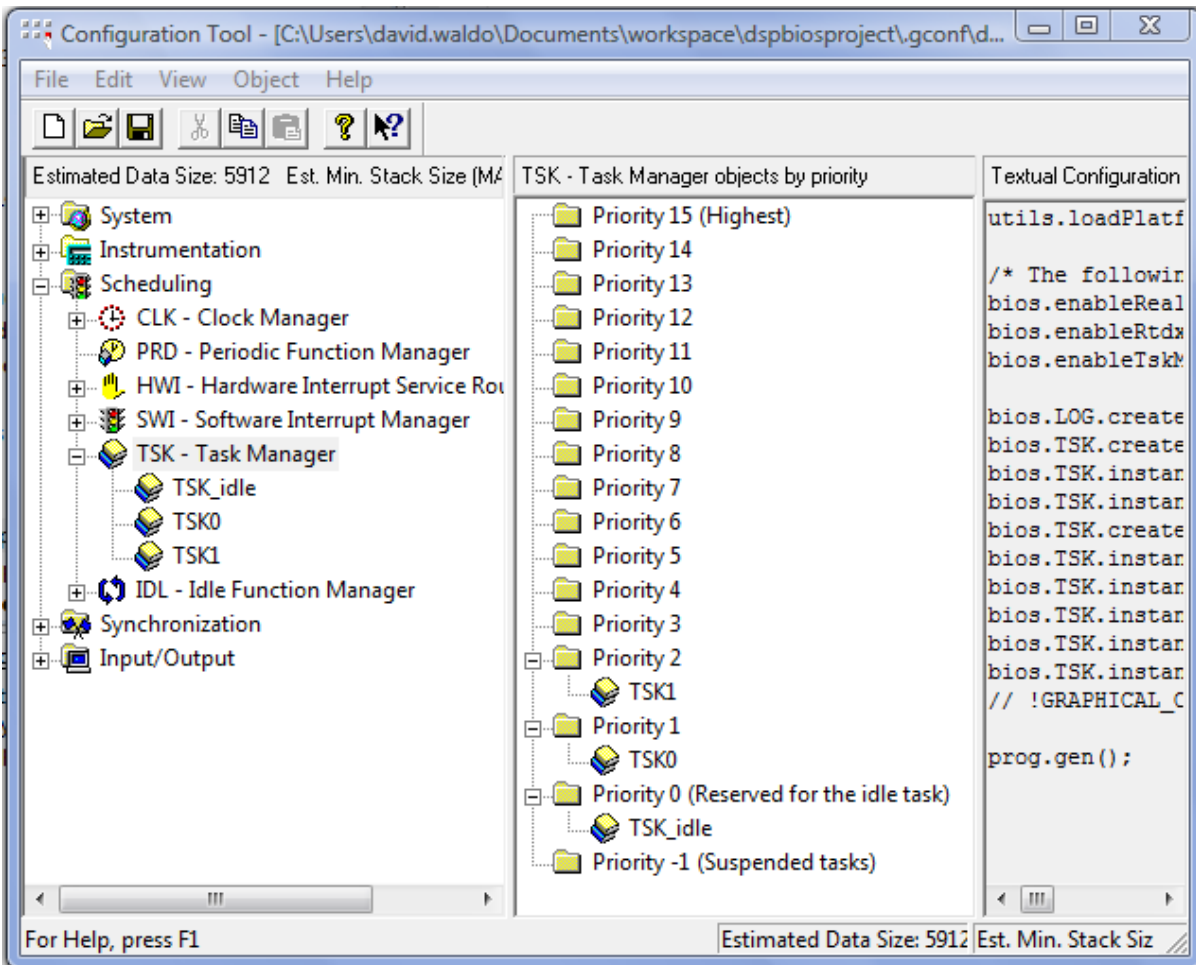


State transition diagram for a task

To create a TSK object, open the configuration file, right click on **Scheduling->TSK** and select **Insert TSK**. You can right click on the object and select **Properties** to change its properties. Set the TSK priority on the General tab. Click on the Function tab and put the function name (preceded by an underscore) in for the function you want to handle the TSK. Figure 2 shows a view of the configuration file where the TSKs can be seen in their

corresponding priority. Notice that the `TSK_idle` is priority 0 and it is reserved. This is the idle task.

When your program runs, the tasks with the same priority will get initialized in the order that they are shown in the configuration tool.



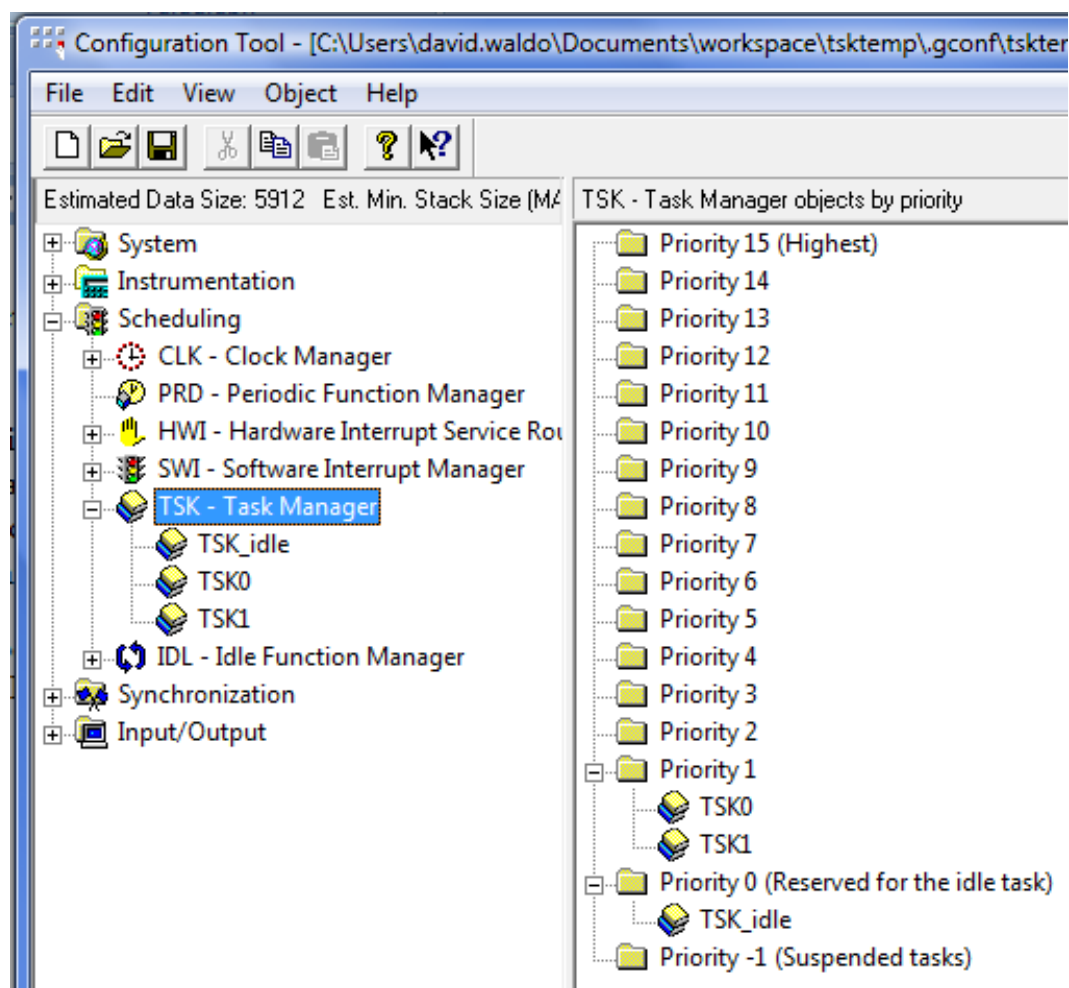
Configuration file showing TSKs and priorities

There are many functions in the TSK module. Some of the more commonly used ones are:

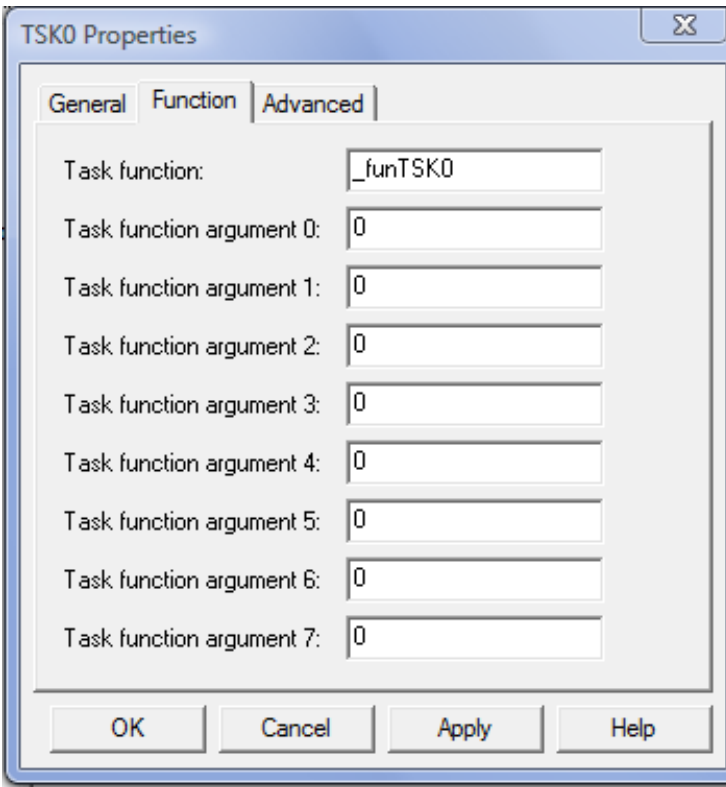
- `TSK_disable` - Disable DSP/BIOS task scheduler

- **TSK\_enable** - Enable DSP/BIOS task scheduler
- **TSK\_yield** - Yield processor to equal priority task
- **TSK\_sleep** - Delay execution of the current task

As an example, suppose there are two task at the same priority as shown in the configuration file in Figure 3. To set up the threads of execution, change the Function properties to **\_funTSK0** for task TSK0 and **\_funTSK1** for task TSK1 as shown in Figure 4.



Configuration tool showing two tasks at Priority 1



Function name for the Task

Then the `main.c` file will need two functions `funTSK0` and `funTSK1` as shown in the following listing.

```
#include <std.h>
#include <log.h>
#include <tsk.h>
#include "tskcfg.h"

Void main()
{
}

Void funTSK0()
{
LOG_printf(&trace, "TSK0 Start");
```

```

TSK_yield();
LOG_printf(&trace, "TSK0 Main");
TSK_yield();
LOG_printf(&trace, "TSK0 Finish");
}

Void funTSK1()
{
LOG_printf(&trace, "TSK1 Start");
TSK_yield();
LOG_printf(&trace, "TSK1 Main");
TSK_yield();
LOG_printf(&trace, "TSK1 Finish");
}

```

When the program starts, the TSK0 runs first since it is the first on the list under Priority 1. It will print one statement and then yield to another task at the same priority, which will be TSK1. TSK1 will then begin its execution. The result of the whole run is:

```

TSK0 Start
  TSK1 Start
  TSK0 Main
  TSK1 Main
  TSK0 Finish
  TSK1 Finish

```

## Viewing Objects

When debugging a project it is important to see the characteristics of the components in your project. When using DSP/BIOS in CCS it is possible to view the DSP/BIOS objects and their properties using the RTSC Object Viewer (ROV). RTSC is Real Time Software Components which is a standard for developing software modules/libraries.



To view the DSP/BIOS v5.x objects and properties in the debug mode, select **Tools->ROV**. This will bring up a window that looks like the following figure.

na...	handle	state	priority	timeout	timeRemaining	blockedOn	stackBase	stackSize	stackPeak
TSK_idle	0x00007ecc	Ready	0	0	0		0x000067f8	1024	168
TSK0	0x00007f2c	Running	1	0	0		0x00005ff8	1024	148
TSK1	0x00007f8c	Ready	1	0	0		0x000063f8	1024	148

RTSC Object Viewer

This figure shows the TSK module and the two tasks in the project. Notice the different properties that can be seen for each task.

## TI DSP/BIOS SEM Module

This module describes the basics of the TI DSP/BIOS Semaphore Module (SEM).

### Introduction

The DSP/BIOS real-time operating system provides for inter-task synchronization through the use of semaphores. The module that provides this function is the SEM module. This module will describe the basic use of SEM objects that are initialized in the configuration tool.

### Reading

- SPRU423 TMS320 DSP/BIOS Users Guide: Read the section on Semaphores.

### SEM Module

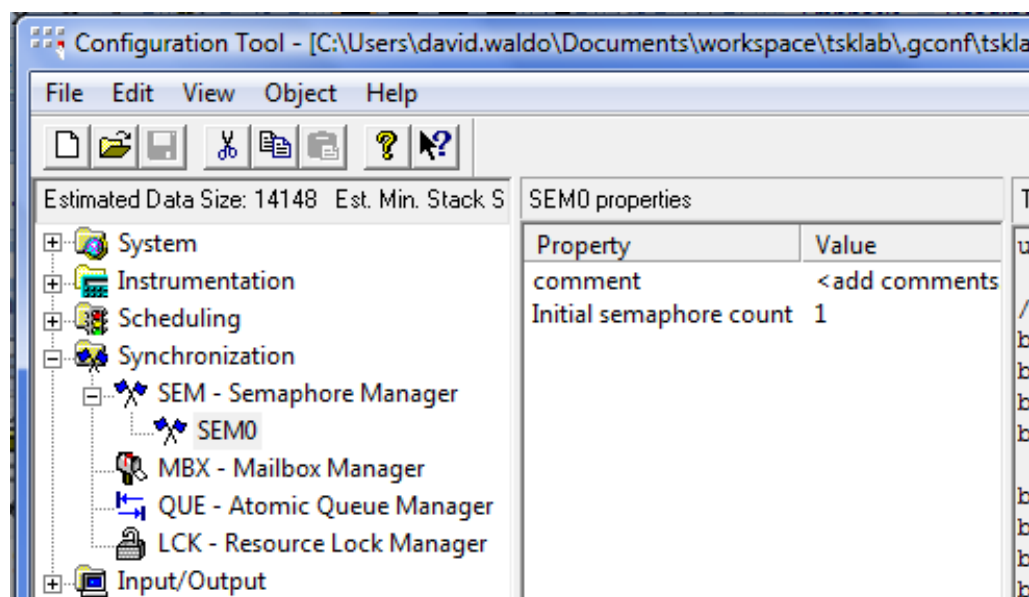
Semaphores provide a way for intertask communication and synchronization. They are used to coordinate access to a shared resource being accessed by multiple tasks. The SEM objects are counting semaphores which means they keep a record of the number of corresponding resources available.

There are two main functions for using a semaphore once it has been created.

- **SEM\_pend** is used to wait for a semaphore. If the semaphore has a positive value, this function will simply decrement the semaphore and return. If the value of the semaphore is 0, the task calling **SEM\_pend** is blocked and its state is changed to **TSK\_BLOCKED**. The timeout parameter to **SEM\_pend** allows the task to wait until a timeout occurs, wait indefinitely (**SYS\_FOREVER**), or not wait at all. **SEM\_pend**'s return value is used to indicate if the semaphore was signaled successfully.

- **SEM\_post** is used to signal a semaphore. If a particular task is waiting for the semaphore (in the **TSK\_BLOCKED** state), **SEM\_post** removes the task from the semaphore queue and puts it on the ready queue (in the **TSK\_READY** state). In this case, the semaphore remains at 0. If no tasks are waiting, **SEM\_post** simply increments the semaphore count and returns.

To create an SEM object, open the configuration file, right click on **Synchronization-> SEM** and select **Insert SEM**. You can right click on the object and select **Properties** to change its properties. The following figure shows a semaphore, SEM0, in the configuration tool with an initial value of 1.



Semaphore in configuration tool

As an example, suppose there are two tasks, TSK0 and TSK1, that are at the same priority. When program execution begins, TSK0 starts first. The semaphore, SEM0, is initialized to 1. The following code is used for the two tasks.

```

funTSK0(){
    1 LOG_printf(&trace, "TSK0 Start");
    2 SEM_pend(&SEM0, SYS_FOREVER);
    3 LOG_printf(&trace, "TSK0 1");
    4 SEM_pend(&SEM0);
    11 LOG_printf(&trace, "TSK0 2");
    12 SEM_post(&SEM0);
    13 SEM_post(&SEM0);
    14 LOG_printf(&trace, "TSK0 3");
    15 SEM_post(&SEM0);
    16 LOG_printf(&trace, "TSK0 Finish");
}
funTSK1(){
    5 LOG_printf(&trace, "TSK1 Start");
    6 SEM_post(&SEM0);
    7 LOG_printf(&trace, "TSK1 1");
    8 SEM_post(&SEM0);
    9 SEM_pend(&SEM0);
    10 SEM_pend(&SEM0);
    17 LOG_printf(&trace, "TSK1 2");
    18 LOG_printf(&trace, "TSK1 Finish");
}

```

The following table shows the order of execution, the state of the two tasks, the value of the semaphore and any output printed. The Code Line # indicates which line of code is executed. The line in the table indicates the result after the line of code is executed. For instance, the line 1 indicates that the following line of code is executed:

```
1 LOG_printf(&trace, "TSK0 Start");
```

In the table, the TSK0 is in the TSK\_RUNNING (E) state, TSK1 is in the TSK\_READY (R) state, the semaphore has a value of 1 and the line causes the output "TSK0 Start". The next line of code to execute is

```
2 SEM_pend(&SEM0, SYS_FOREVER);
```

The other states for the tasks are TSK\_BLOCKED (B) and TSK\_TERMINATED (T).

Code Line #	TSK0 State	TSK1 State	SEM0	Output
1	E	R	1	TSK0 Start
2	E	R	0	
3	E	R	0	TSK0 1
4	B	E	0	
5	B	E	0	TSK1 Start
6	R	E	0	
7	R	E	0	TSK 1
8	R	E	1	
9	R	E	0	
10	E	B	0	
11	E	B	0	TSK0 2

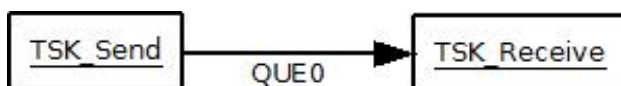
12	E	R	0	
13	E	R	1	
14	E	R	1	TSK0 3
15	E	R	2	
16	E	R	2	TSK0 Finish
17	T	E	2	TSK1 2
18	T	T	2	TSK1 Finish

## Semaphore as a resource monitor

There are a couple common practical uses for semaphores. One of those uses is to count how many items are on a queue. The other is to restrict tasks from accessing a resource at the same time.

### Queue count

Suppose there are two tasks in your system, TSK\_Send and TSK\_Receive and there is one queue between them, QUE0. To keep track of how many elements are on the queue there is a counting semaphore, SEM0. The semaphore is initialized to 0. See Figure 1.



Tasks and queue

When TSK\_Send puts a message on the queue, it will perform a `SEM_post(&SEM0)` which will increment the count on the semaphore. Before the TSK\_Receive tries to get a message off the queue it will perform a `SEM_pend(&SEM0, SYS_FOREVER)`. This will decrement the count on the semaphore every time the TSK\_Receive takes a message off the queue. If there are no messages on the queue, SEM0 will be 0 and the task will block. This will cause the task to wait until the TSK\_Send task puts another message on the queue.

## Resource sharing

When two tasks share a resource, it may be necessary for the tasks have mutually exclusive access to the resource. In this case, a semaphore can be used to acquire access to the resource or lock access to a resource.

Suppose there are two tasks with the same priority in your system, TSK0 and TSK1. There is a Resource that the two tasks share. A semaphore, SEM0, will be used to acquire access to the resource. The semaphore is initialized to 1. See Figure 2.



Two tasks sharing a resource

In order for the tasks to have mutually exclusive access to the resource, each task must perform a `SEM_pend(&SEM0, SYS_FOREVER)` before using the resource and a `SEM_post(&SEM0)` after using the resource.

Examine the following code that does NOT use a semaphore when writing to the resource.

```
int Resource[4];
Void funTSK0()
{
    Resource[0]=0;
    Resource[1]=1;
    TSK_yield();
    Resource[2]=2;
    Resource[3]=3;
}

Void funTSK1()
{
    Resource[0]=4;
    Resource[1]=5;
    Resource[2]=6;
    Resource[3]=7;
}
```

When the tasks finish the contents of the array Resource will be [4,5,2,3]. Clearly, we would like it to be either [0,1,2,3] or [4,5,6,7]. To fix this, add the semaphore.

```
int Resource[4];
Void funTSK0()
{
    SEM_pend(&SEM0, SYS_FOREVER);
    Resource[0]=0;
    Resource[1]=1;
    TSK_yield();
    Resource[2]=2;
    Resource[3]=3;
    SEM_post(&SEM0);
}
```



```

Void funTSK1()
{
    SEM_pend(&SEM0, SYS_FOREVER);
    Resource[0]=4;
    Resource[1]=5;
    Resource[2]=6;
    Resource[3]=7;
    SEM_post(&SEM0);
}

```

When the tasks finish the contents of the array Resource will be [4,5,6,7]. The reason for this is that after TSK0 performs a **TSK\_yield**, TSK1 tries to perform a **SEM\_pend** but gets blocked. At that point TSK0 continues running and finishes with the resource. Then TSK1 begins to run and writes the complete array.

## Viewing Objects

When debugging a project it is important to see the characteristics of the components in your project. When using DSP/BIOS in CCS it is possible to view the DSP/BIOS objects and their properties using the RTSC Object Viewer (ROV). RTSC is Real Time Software Components which is a standard for developing software modules/libraries.

To view the DSP/BIOS v5.x objects and properties in the debug mode, select **Tools->ROV**. This will bring up a window that looks like the following figure.

Problems RTA Control Panel ROV Printf Logs

LCK\_test.out

Viewable Modules

KNL

LOG

MBX

MEM

SEM

SWI

TSK

Instances

name	handle	count	numTasksPending	pendQ
SEM0	0x0000858c	1	0	

RTSC Object Viewer

This figure shows the SEM module and the semaphore in the project. Notice the different properties that can be seen for each semaphore.

## TSK SEM Lab

The module is a lab assignment to better understand the TI DSP/BIOS TSK and SEM modules.

## Introduction

This lab module will help you become familiar with the TI DSP/BIOS TSK module and SEM module.

## Reading

- SPRU423 TMS320 DSP/BIOS Users Guide: Read the sections on Semaphores and Tasks.

## Lab Module Prerequisites

None

## Laboratory

### Part 1

- In this part we will create two tasks that will have the same priority and will run independent of each other.
- Follow the procedure in [Code Composer Studio v4 DSP/BIOS Project](#) to create a new CCS DSP/BIOS project. Name the project **tsklab**. Your project should have a DSP/BIOS configuration file named **tsklab.tcf**.
- Create a LOG object by right clicking on Instrumentation->LOG and selecting Insert LOG. Change the name to **trace**. Set its properties to have a length of 128 and be a fixed buffer.
- Create two TSKs with the following properties
  1. Name: **TSK0**, priority: 1, function: **\_funTSK0**.
  2. Name: **TSK1**, priority: 1, function: **\_funTSK1**.

- Create a `main.c` file and include a `main` function that does nothing.
- In the `main.c` file make functions for your TSKs, `funTSK0` and `funTSK1`. Each function should print to the `trace` LOG object once at the beginning of the function to indicate that the particular TSK is starting and once at the end of the function to indicate that the particular TSK is ending.
- Add code to each task that loops 5 times and prints to `trace` the task number and the loop number each time through the loop. It should look something like:

```
TSK0 Number 0
    TSK0 Number 1
    ...
```

- The basic structure of `main.c` should be:

```
#include <std.h>
#include "tsklabcfg.h"

void main()
{
}

void funTSK0()
{
/* code here */
}

void funTSK1()
{
/* code here */
}
```

- Start the debug session.
- Open the log view by selecting **Tools->RTA->Printf Logs**.

- Run the program and record the results.

## Part 2

- In this part we will change the tasks so that they yield to each other within their loops.
- Have in your system two TSKs with the following properties
  1. Name: **TSK0**, priority: 1, function: **\_funTSK0**.
  2. Name: **TSK1**, priority: 1, function: **\_funTSK1**.
  3. Each yields to the other in their loop.
- Copy the **main.c** file from above to a new file **main2.c**. Exclude the **main.c** from your project build.
- Change the code so that the two tasks yield to each other within the loop. Put a print statement before the yield and one after the yield so you will know when the function has completed the loop. Do this for subsequent parts of this lab also.
- Start the debug session.
- Open the log view by selecting **Tools->RTA->Printf Logs**.
- Run the program and record the results.

## Part 3

- In this part we will change the tasks so that one is a higher priority than the other. They yield to each other in their loops.
- Have in your system two TSKs with the following properties
  1. Name: **TSK0**, priority: 2, function: **\_funTSK0**.
  2. Name: **TSK1**, priority: 1, function: **\_funTSK1**.
  3. Each yields to the other in their loop.
- This part will still be executing **main2.c**.
- Set **TSK0** to a higher priority than the other.

- Start the debug session.
- Open the log view by selecting **Tools->RTA->Printf Logs**.
- Run the program and record the results.
- Does the execution appear different from part 2?

## Part 4

- In this part you will add a semaphore to the program. One task will pend on the semaphore and the other will post to the semaphore.
- Have in your system two TSKs with the following properties
  1. Name: **TSK0**, priority: 1, function: **\_funTSK0**.
  2. Name: **TSK1**, priority: 1, function: **\_funTSK1**.
  3. **TSK0** pends on the semaphore and **TSK1** posts to the semaphore in the loop.
- Have in your system a semaphore with the following properties
  1. Name: SEM0, initial value: 1
- Name your new **main** file **main4.c**.
- Remove the task yield from each task.
- Use **SYS\_FOREVER** for the timeout on the pend.
- Put **TSK0** first in the priority list in the configuration file. This will cause it to run first.
- Start the debug session.
- Open the log view by selecting **Tools->RTA->Printf Logs**.
- Run the program and record the results.

## Part 5

- In this part you change the initial value on the semaphore and observe what happens.
- Have in your system two TSKs with the following properties

1. Name: **TSK0**, priority: 1, function: **\_funTSK0**.
2. Name: **TSK1**, priority: 1, function: **\_funTSK1**.
3. **TSK0** pends on the semaphore and **TSK1** posts to the semaphore.

- Semaphore with the following properties

1. Name: SEM0, initial value: 2, 3 (run multiple times to get these).

- Change the semaphore initial value to 2.
- Start the debug session.
- Open the log view by selecting **Tools->RTA->Printf Logs**.
- Run the program and record the results.
- Run the program with different SEM0 initial values.
- What happens with the different initial values?

## Part 6

- In this part you will make the task that posts to the semaphore a higher priority.
- Two TSKs with the following properties
- Have in your system two TSKs with the following properties

1. Name: **TSK0**, priority: 1, function: **\_funTSK0**.
2. Name: **TSK1**, priority: 2, function: **\_funTSK1**.
3. **TSK0** pends on the semaphore and **TSK1** posts to the semaphore.

- Semaphore with the following properties

1. Name: SEM0, initial value: 1

- Make the task that is posting a higher priority than the other.
- Start the debug session.
- Open the log view by selecting **Tools->RTA->Printf Logs**.
- Run the program and record the results.

## Part 7

- In this part you will make the task that pends on the semaphore a higher priority.
- Have in your system two TSKs with the following properties
  1. Name: **TSK0**, priority: 2, function: **\_funTSK0**.
  2. Name: **TSK1**, priority: 1, function: **\_funTSK1**.
  3. **TSK0** pends on the semaphore and **TSK1** posts to the semaphore.
- Semaphore with the following properties
  1. Name: SEM0, initial value: 2
- Start the debug session.
- Open the log view by selecting **Tools->RTA->Printf Logs**.
- Run the program and record the results.

## Part 8

- In this part you will run the code that is given for each task and then document for each step through the code the state of the tasks and semaphore and any output that is printed.
- Have in your system two TSKs with the following properties
  1. Name: **TSK0**, priority: 1, function: **\_funTSK0**.
  2. Name: **TSK1**, priority: 1, function: **\_funTSK1**.
- Put **TSK0** first in the priority list in the configuration file. This will cause it to run first.
- Semaphore with the following properties
  1. Name: SEM0, initial value: 1
- Use the following code for your tasks.



```

void funTSK0()
{
    LOG_printf(&trace, "TSK0 Starting");
    SEM_post(&SEM0);
    LOG_printf(&trace, "TSK0 1");
    SEM_pend(&SEM0, SYS_FOREVER);
    LOG_printf(&trace, "TSK0 2");
    SEM_pend(&SEM0, SYS_FOREVER);
    SEM_pend(&SEM0, SYS_FOREVER);
    LOG_printf(&trace, "TSK0 3");
    SEM_post(&SEM0);
    LOG_printf(&trace, "TSK0 Finish");
}
funTSK1(){
    LOG_printf(&trace, "TSK1 Start");
    SEM_post(&SEM0);
    LOG_printf(&trace, "TSK1 1");
    SEM_pend(&SEM0, SYS_FOREVER);
    SEM_post(&SEM0);
    SEM_pend(&SEM0, SYS_FOREVER);
    LOG_printf(&trace, "TSK1 2");
    LOG_printf(&trace, "TSK1 Finish");
}

```

- Fill out the following table as you run your program.

Code Line #	TSK0 State	TSK1 State	SEM0	Output
1				

2				
3				
4				
5				
6				
7				
8				
9				
10				
11				
12				
13				
14				
15				
16				
17				
18				

## Laboratory Write-up

- For each part of this lab, explain the setup of the objects and what they are doing. Explain the results in great detail showing values for things like the semaphore count and why the task is running.

## TI DSP/BIOS QUE Module

This module describes the basics of the TI DSP/BIOS v5.x Queue (QUE) Module.

### Introduction

This module describes the basics of the Queue (QUE) module in DSP/BIOS v5.x. It also shows how to use the MEM module to allocate memory for queue elements.

### Reading

- SPRU423 TMS320 DSP/BIOS Users Guide: Read the section title Queues
- SPRU403 TMS320C6000 DSP/BIOS 5.x Application Programming Interface (API) Reference Guide: Reference the sections on QUE and MEM

### QUE Module

The QUE module provides a way to manage linked lists of data objects. A linked list is a group of objects where an object in the list has information about the previous and next elements in the list as demonstrated in Figure 1. A linked list can be used to implement buffers that operate in a FIFO, LIFO, LILO and FILO manner, to name a few. Many times it is used in a FIFO manner.

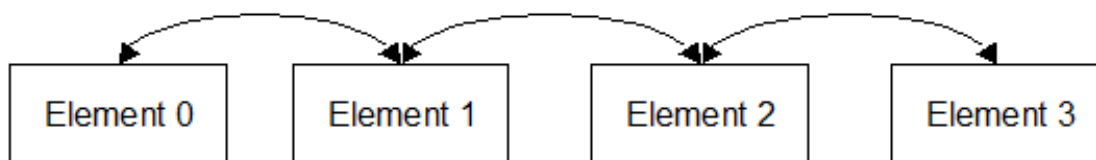


Diagram of a linked list

The basic element in a queue structure is a `QUE_Elem` which is defined by

```
typedef struct QUE_Elem {  
    struct QUE_Elem *next;  
    struct QUE_Elem *prev;  
} QUE_Elem;
```

This definition is in the header file `que.h` which must be included in the source file.

A message or object put on a queue is a structure that has as its first element a `QUE_Elem`. The message is defined by the user's code. An example would be:

```
typedef struct MsgObj {  
    QUE_Elem elem; /* first field for QUE */  
    Int val; /* message value */  
} MsgObj, *Msg;
```

To put a message at the end of a queue use the function `QUE_put` and to remove a message from the beginning of the queue use `QUE_get`. These two functions allow the queue to be used as a First In First Out (FIFO) buffer.

Here is another example of a message structure. Notice that you can put anything else in the structure for your message as long as the `QUE_Elem` element is the first in the structure.

```
typedef struct MsgObj {  
    QUE_Elem elem; /* first field for QUE */  
    float array[100]; /* message array */  
} MsgObj, *Msg;
```

When creating messages it may be necessary to get a segment of memory to store the message. The following code can be used to allocate memory for a message and put it on a queue.

```
Msg msg; // pointer to a MsgObj, this doesn't
allocate memory for a message

    msg = MEM_alloc(0, sizeof(MsgObj), 0); //
Allocate memory for the object
    if (msg == MEM_ILLEGAL) {
        SYS_abort("Memory allocation failed!\n");
    }
    /* fill the message with data here */
    /* This puts the address of the message in the
queue */
    QUE_put(&queue, msg);
```

In order to use the MEM module the header file `mem.h` must be included in the source file. The `MEM_alloc` gets a segment of memory from the memory manager and returns a pointer to the segment. The address gets put into the `msg` pointer so that when we access the object it points to segment of memory.

`QUE_put` puts the address of the message object on the queue. This function only copies the message pointer in order to minimize the amount of processing needed to put the message on the queue.

Before trying to get a message from a queue, the code should check to see if there is a message available on the queue. The following code makes this check and then gets the message from the queue.

```
if (QUE_empty(&queue)) { // Check to see if the
queue is empty
    LOG_printf(&trace, "queue error\n"); // Print
something if it is empty
    // If the queue is empty you probably will not
want to proceed
```

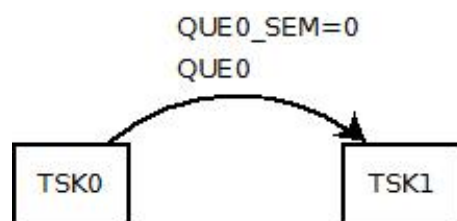
```

    }
    msg = QUE_get(&queue); // If there is a
message, dequeue it
    // use the message here
    // After using the message, free the memory
    // This tells MEM_free the location of the msg
and size so it
    // can remove it from memory
    MEM_free(0, msg, sizeof(MsgObj)); // Free up
the memory

```

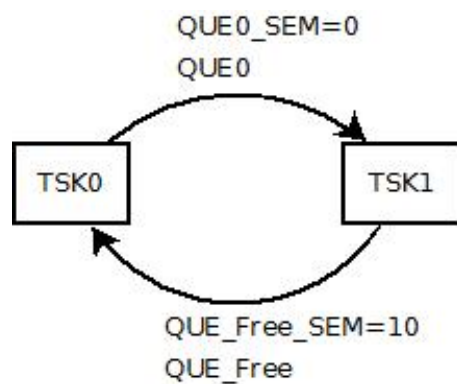
Functions `QUE_put` and `QUE_get` are atomic in that they add and remove elements from the queue with interrupts turned off. Therefore there should not be a problem of more than one task trying to access the queue at the same time. The function `QUE_get` is also non-blocking so the tasks should determine if there are any elements on the queue before calling `QUE_get`.

A semaphore can be used to count the number of elements on a queue and be used to block a task that needs access to a queue. Figure 2 shows how one task `TSK0` will write to a queue, `QUE0`, and the other task `TSK1` will read from the queue. Also, the semaphore `QUE0_SEM` is used to keep track of how many elements are on the queue. After `TSK0` puts a message on the queue it will call `SEM_post` and increment the semaphore. Before reading from the queue, `TSK1` will call `SEM_pend` on the semaphore and if the task does not block, there is an element on the queue. If there are no elements on the queue, the task will block on the semaphore.



Queue and semaphore  
setup

For the setup with one queue and one semaphore notice that if the program ran for a long time the code would have to continually allocate memory for a message and then de-allocate it when it was done using the message. This could take up a substantial amount of time and could cause fragmentation of the memory space. A better method is to have two queues where one queue holds messages that are free and one holds messages that contain data being transmitted from one task to another. Figure 3 shows the same setup in Figure 2 except there is now a queue that contains free messages or empty messages.



Free message queue  
setup

During the initialization phase of the program, memory for empty messages is allocated and the messages are put in the **QUE\_Free** queue and the semaphore **QUE\_Free\_SEM** is incremented for each message put on the queue. When **TSK0** needs to send a message to **TSK1** it will check **QUE\_Free\_SEM** to see if there are any free messages. If not, it will block. If there are free messages it will take one off of **QUE\_Free** after the **QUE\_Free\_SEM** is decremented and then fill the message with data and



put it on the queue **QUE0**. The semaphore **QUE0\_SEM** is incremented after the message is put on **QUE0**.

The task **TSK1** will block on **QUE0\_SEM** until task **TSK1** puts a message on the queue. Then it will decrement the semaphore and use the message. When it is done it will put the message on **QUE\_Free** and increment its semaphore.

## Example

This is a very simple example to demonstrate the structure of a program that uses queues. In the example we will assume that DSP/BIOS has been set up with two tasks, TSK0 and TSK1, and one queue, QUE0. The two tasks have the same priority and TSK0 is set to execute first. The code for the example follows.

```
#include <std.h> // Target definition header
#include <sys.h> // DSP/BIOS config/error
header
#include <log.h> // LOG module header
#include <mem.h> // MEM module header
#include <que.h> // QUE module header
#include <tsk.h> // TSK module header

#include "QUE_Examplecfg.h" // header
generated by QUE_Example.tcf config file

typedef struct MsgObj {
    QUE_Elem elem; /* first field for QUE */
    Int val; /* message value */
} MsgObj, *Msg;
Void main()
{
}
// TSK0 will generate two messages and put
them on the queue
Void funTSK0()
```

```

{
Msg msg; // Pointer to the message object
// allocate memory for first message
msg = MEM_alloc(0, sizeof(MsgObj), 0);
if (msg == MEM_ILLEGAL) {
// If the memory allocation fails, abort
SYS_abort("Memory allocation failed!\n");
}
msg->val = 1; // put the message number in the
message
// print the message number
LOG_printf(&trace, "Writing message %d", msg-
>val);
// Put the message on the queue
QUE_put(&QUE0, msg);

// repeat for the second message
msg = MEM_alloc(0, sizeof(MsgObj), 0);
if (msg == MEM_ILLEGAL) {
// If the memory allocation fails, abort
SYS_abort("Memory allocation failed!\n");
}
msg->val = 2; // put the message number in the
message
// print the message number
LOG_printf(&trace, "Writing message %d", msg-
>val);
// Put the message on the queue
QUE_put(&QUE0, msg);
}
// TSK1 will get two messages from the queue
Void funTSK1()
{
Msg msg; // Pointer to the message object

// If the queue is empty, we should not

```

```

proceed
    if (QUE_empty(&QUE0)) {
        LOG_printf(&trace, "TSK1 queue error");
        return; // This will make the task terminate
    }
    // Get the message off the queue
    msg = QUE_get(&QUE0);
    // print value in the message
    LOG_printf(&trace, "Reading message %d", msg-
>val);
    // Since we are done with the message, free
the memory
    MEM_free(0, msg, sizeof(MsgObj));

// Repeat for the second message
    // If the queue is empty, we should not
proceed
    if (QUE_empty(&QUE0)) {
        LOG_printf(&trace, "TSK1 queue error");
        return; // This will make the task terminate
    }
    // Get the message off the queue
    msg = QUE_get(&QUE0);
    // print value in the message
    LOG_printf(&trace, "Reading message %d", msg-
>val);
    // Since we are done with the message, free
the memory
    MEM_free(0, msg, sizeof(MsgObj));
}

```

It is important to free the memory of each message after it is used so that the memory does not get used up. The result of the run follows.

Writing message 1  
Writing message 2  
Reading message 1  
Reading message 2

## QUE Lab

This module is a lab assignment to better understand the TI DSP/BIOS QUE module.

## Introduction

This lab module will help you become familiar with the TI DSP/BIOS v5.x QUE module. Code Composer Studio v4 is used in this module.

## Reading

- SPRU423 TMS320 DSP/BIOS Users Guide: Read the section title Queues

## Lab Module Prerequisites

This lab module uses aspects of the TSK and SEM modules.

## Laboratory

### Part 1

- In this part you will be creating two TSKs where one will generate some data and send it to the other TSK in a queue. The transmitting TSK will generate 5 messages to send to the receiving TSK which will print out the content of the message.
- Follow the procedure in [Code Composer Studio v4 DSP/BIOS Project](#) to create a new CCS DSP/BIOS project. Your project should have a DSP/BIOS v5.x configuration file using the ti.platforms.sim67xx template (or another appropriate template).
- Create a LOG object by right clicking on Instrumentation->LOG and selecting Insert LOG. Change the name to **trace**. Set its properties to have a length of 512 and be a fixed buffer.
- Change the LOG\_system object to have a length of 512.

- If using the simulator then change the RTDX interface to Simulator by right clicking on Input/Output->RTDX and bringing up the properties. Change the RTDX mode to Simulator. If you do not do this then when you load your program you will see the error **RTDX application does not match emulation protocol**. If you are loading onto an EVM or DSK you leave the setting on JTAG.
- Create a QUE object by right clicking on Synchronization->QUE and selecting Insert QUE. This will create a queue called **QUE0**. There are no properties to set (unless you want to change the comment).
- Create two TSKs with the following properties
  - Name: **TSK0**, priority: 1, function: **\_funTSK0**.
  - Name: **TSK1**, priority: 1, function: **\_funTSK1**.
- On the priority list make sure that **TSK0** is first.
- Create a **main.c** file and include a **main** function that does nothing.
- At the top of **main.c** make a global structure for your message as follows:

```
typedef struct MsgObj {
    QUE_Elem elem; /* first field for QUE */
    Int val; /* message value */
} MsgObj, *Msg;
```

- In the **main.c** file make functions for your TSKs, **funTSK0** and **funTSK1**.
- In **funTSK0**, which will generate the data, create a loop that will loop 5 times and in the loop have it:
  1. Allocate memory for a new message
  2. Fill the message value with the message number. Use a command like: **"msg->val = ..."**
  3. Print to the trace LOG which message is being generated
  4. Put the message on the queue QUE0

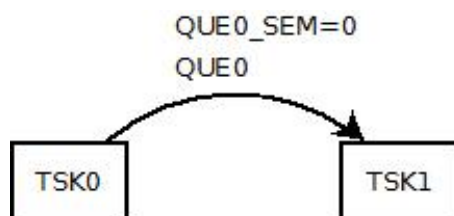
- In **funTSK1**, which will receive the data, create a loop that will loop 5 times and in the loop have it:
  1. Check to see if there is a message on the queue and if not then print an error message to the **trace** LOG and **return**. This will cause the task to be done running and enter the terminated state.
  2. Get the message from the queue **QUE0**
  3. Print which message number was read
  4. Free the buffer that was used for the message
- Start the debug session.
- Open the log view by selecting **Tools->RTA->Printf Logs**.
- Run the program and record the results.
- Describe in detail the processing that is occurring.

## Part 2

- Change the priority of **TSK1** to 2.
- Start the debug session.
- Run the program and record the results.
- How does the execution change? Describe in detail.

## Part 3

- In this part you will be adding a semaphore to synchronize the two tasks using the queue. The semaphore will keep track of how many messages are in the queue.



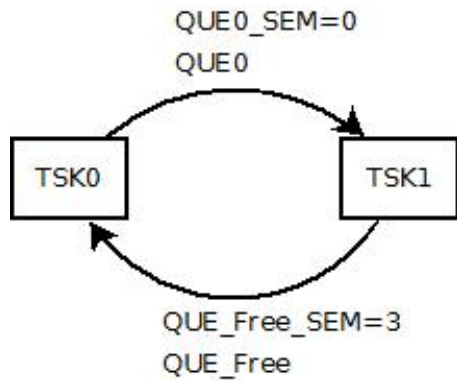
## Setup for Part 3

- Copy the `main.c` file from above to a new file `main3.c`. Remove the `main.c` from your project and add `main3.c`.
- Add a semaphore to your configuration file and call it `QUE0_SEM`. It should be initialized to 0.
- Keep the priority of `TSK0` set to 1 and `TSK1` set to 2.
- In `main3.c` change the following:
  1. In `funTSK0` after the `QUE_put` add a `SEM_post` for `QUE0_SEM`.
  2. In `funTSK1` delete the code that checks to see if there is a message on the queue and replace it with a `SEM_pend` for `QUE0_SEM`. Use `SYS_FOREVER` for the timeout on the pend.
- Start the debug session.
- Run the program and record the results.
- Describe in detail the processing that is occurring.

## Part 4

- Notice that in Part 3 that if the program ran for a long time that the code would have to continually allocate memory for a message and then de-allocate it when it was done using the message. This could take up a substantial amount of time and could cause fragmentation of the memory space. A better method is to have two queues where one queue holds messages that are free and one holds messages that contain data being transmitted from one task to another. In this part you will be adding another queue which will be initialized by adding some free messages to it. Also, another semaphore will be added to track the number of free messages on the free queue.





Setup for Part 4 with a  
free message queue

- Copy the `main3.c` file from above to a new file `main4.c`. Remove the `main3.c` from your project and add `main4.c`.
- Add another semaphore to your configuration file and call it `QUE_Free_SEM`. It should be initialized to 0.
- Create a QUE object and call it `QUE_Free`.
- Change the priority of `TSK1` to 1 and make sure `TSK0` is first in the list.
- To the `main` function add code that loops 3 times and adds 3 messages to the queue `QUE_Free`. You will need to allocate memory for the messages. Be sure to post to `QUE_Free_SEM` after adding the message to the queue. This will set up the free messages to be used in the program. These messages can then be used and reused without having to allocate and free memory over and over again.
- In the function `funTSK0` delete the code that allocates a message and replace it with code that pends on `QUE_Free_SEM` and then gets a message from `QUE_Free`.
- In the function `funTSK1` delete the code that frees the message buffer and replace it with code that puts the message on `QUE_Free` and then posts to `QUE_Free_SEM`.
- Start the debug session.
- Run the program and record the results.
- Describe in detail the processing that is occurring.

## Microsoft Visual Basic Tasks and Semaphores

This module describes the basics of Microsoft Visual Basic Threads and Semaphores.

### Introduction

Microsoft Visual Basic provides resources for managing threads and semaphores. This module will give a basic overview of threads and semaphores in Visual Basic (VB). This module presents threads and semaphores in VB in a very simplistic manner. There are much better ways to use threads in VB but they will be presented here in a way that makes the programs much easier to write. The threads described here will be methods of the Form class. This is what makes the programming very simple. Threads should/could be classes on their own but it is not necessary. This module is not intended to make you proficient in using threads and semaphores in Visual Basic.

### Reading

- System.Threading Namespace: <http://msdn.microsoft.com/en-us/library/system.threading.aspx>
- Semaphore Class: <http://msdn.microsoft.com/en-us/library/system.threading.semaphore.aspx>
- Thread Class: <http://msdn.microsoft.com/en-us/library/system.threading.thread.aspx>
- Console WriteLine method: <http://msdn.microsoft.com/en-us/library/586y06yf.aspx>

### Module Prerequisites

It would be helpful to have completed the TSK and SEM modules for the DSP/BIOS operating system but it is not necessary if you are familiar with tasks and semaphores.

### Visual Basic Threads

A thread is simply a separate method or function that gets executed apart from the main program flow. There can be many threads executing simultaneously in a system. In VB the `System.Threading` namespace needs to be imported into the program. This is done by putting the following at the very top of your code:

```
Imports System.Threading
```

If this is not included at the top of your program then it will not know what threads are.

The following is the basic structure of your program for this lab. Each part of the example code is described in the comments.

```
' This imports the threading namespace so the
program knows about threads
Imports System.Threading

' This is the class that executes the form for
the program
Public Class Form1
' This is a definition of a thread. The thread
object is thread1 and
' it uses the method Task1() below as its
code.
Dim thread1 As New Thread(AddressOf Task1)

' When the form program starts or is loaded
the following function is
' called. This is where you can put the
commands to set up our program.
' The code for starting up the threads can be
put here.
Private Sub Form1_Load(ByVal sender As
System.Object, ByVal e As System.EventArgs)
Handles MyBase.Load
```

```

        ' Make this a background thread so it
automatically
        ' aborts when the main program stops.
thread1.IsBackground = True
        ' Set the thread priority
thread1.Priority = ThreadPriority.Lowest
        ' This command starts the thread. The method
Task1() then
        ' starts to execute.
thread1.Start()
End Sub

Private Sub Task1()
    ' This is where your code goes for this thread
End Sub

End Class

```

The real-time operating system in Visual Studio is not a deterministic operating system. This means that you cannot be sure of how the threads will execute. Windows assigns time to threads at the same priority in a round-robin fashion. When threads at a particular level do not need to execute the operating system assigns time to lower priority threads.

## Visual Basic Semaphores

Semaphores in VB are used to control access to a resource. They act as a counting semaphore except that there is a maximum value that the semaphore can have. This value is set when the semaphore is declared. The following is an example of the definition of a semaphore.

```
Dim Sem As New Semaphore(1, 2)
```

This defines a semaphore object named **Sem** that has a maximum value of 2 and an initial value of 1. The main methods for using the semaphore are

`WaitOne()` and `Release()`. The `WaitOne()` method decrements the semaphore and the `Release()` method increases it. If the value of the semaphore is 0 when the `WaitOne()` method is called the thread will block. When the `Release()` method is called when a thread is blocked on the semaphore, it will unblock a thread without incrementing the semaphore. If the `Release()` method is called too many times and the semaphore value is increased past the maximum value, a `SemaphoreFullException` exception will occur. This module does not discuss how to handle the exception.

The main way to use the semaphore is shown in the following example.

```
Dim Sem As New Semaphore(1, 1)

    Sem.WaitOne()
    ' Use the protected resource here
    Sem.Release()
```

## Writing to the console

To write to the console use function calls such as

```
Console.WriteLine("Name = {0}, hours = {1:hh}",  
My.Name, DateTime.Now)
```

The fixed text is `"Name = "` and `", hours = "`. The format items are `"{0}"`, whose index is 0, which corresponds to the object `My.Name`, and `"{1:T}"`, whose index is 1, which corresponds to the object `DateTime.Now`. The format `T` will print the time in the long time format. An example of the output is:

```
Name = Form1, Time = 5:50:54 PM
```

## Example

This simple example uses Visual Studio 2010 and VB.NET Framework 4. Create a Form Application and enter the following code:

```
Imports System.Threading
Public Class Form1

    Dim thread1 As New Thread(AddressOf Task)
    Dim thread2 As New Thread(AddressOf Task)
    Dim Sem As New Semaphore(1, 1)

    ' When the form program starts or is loaded
    the following function is
    ' called. This is where you can put the
    commands to set up our program.
    Private Sub Form1_Load(ByVal sender As
    System.Object, ByVal e As System.EventArgs)
    Handles MyBase.Load

        ' Make this a background thread so it
    automatically
        ' aborts when the main program stops.
        thread1.IsBackground = True
        ' Set the thread priority to lowest
        thread1.Priority = ThreadPriority.Lowest
        ' Start the thread
        thread1.Start("thread1")

        ' Make this a background thread so it
    automatically
        ' aborts when the main program stops.
        thread2.IsBackground = True
        ' Set the thread priority to lowest
        thread2.Priority = ThreadPriority.Lowest
        ' Start the thread
        thread2.Start("thread2")
```

```

End Sub

Private Sub Task(ByVal name As Object)

While 1
    ' wait on the semaphore
    Sem.WaitOne()
    ' Print the name of this thread and the
current time
    Console.WriteLine("Name = {0}, Time1 = {1:T}",
name, DateTime.Now)
    ' Sleep to 2 seconds
    Thread.Sleep(2000)
    Console.WriteLine("Name = {0}, Time2 = {1:T}",
name, DateTime.Now)
    ' Release the semaphore
    Sem.Release()

End While

End Sub

End Class

```

This program will start up two threads, `thread1` and `thread2`. Each thread will execute the method `Task()` and will pass the name of the thread. This will allow each thread to print something different even though they execute the same method. A semaphore is use to prevent the other thread from interrupting the two print statements. The two print statements for the thread should always print after each other. The following shows the results of the run. Note that the two times always print together.

```

Name = thread1, Time1 = 6:17:16 PM
    Name = thread1, Time2 = 6:17:18 PM
    Name = thread1, Time1 = 6:17:18 PM

```

```
Name = thread1, Time2 = 6:17:20 PM
Name = thread1, Time1 = 6:17:20 PM
Name = thread1, Time2 = 6:17:22 PM
Name = thread2, Time1 = 6:17:22 PM
Name = thread2, Time2 = 6:17:24 PM
Name = thread2, Time1 = 6:17:24 PM
Name = thread2, Time2 = 6:17:26 PM
Name = thread2, Time1 = 6:17:26 PM
Name = thread2, Time2 = 6:17:28 PM
```

The following shows the loop with the semaphore statements removed.

```
While 1
    ' Print the name of this thread and the
current time
    Console.WriteLine("Name = {0}, Time1 = {1:T}",
name, DateTime.Now)
    ' Sleep to 2 seconds
    Thread.Sleep(2000)
    Console.WriteLine("Name = {0}, Time2 = {1:T}",
name, DateTime.Now)
End While
```

This causes a printout like the following. Notice the two prints do not occur next to each other.

```
Name = thread2, Time1 = 6:20:44 PM
    Name = thread1, Time1 = 6:20:44 PM
    Name = thread2, Time2 = 6:20:46 PM
    Name = thread1, Time2 = 6:20:46 PM
    Name = thread1, Time1 = 6:20:46 PM
    Name = thread2, Time1 = 6:20:46 PM
    Name = thread2, Time2 = 6:20:48 PM
    Name = thread1, Time2 = 6:20:48 PM
    Name = thread1, Time1 = 6:20:48 PM
    Name = thread2, Time1 = 6:20:48 PM
```



Name = thread2, Time2 = 6:20:50 PM  
Name = thread2, Time1 = 6:20:50 PM  
Name = thread1, Time2 = 6:20:50 PM  
Name = thread1, Time1 = 6:20:50 PM

## Visual Basic Threads and Semaphores Lab

The module is a lab assignment to help you better understand the very basics of Microsoft Visual Basic threads and semaphores.

### Introduction

This lab module will help you become familiar with the basics of threads and semaphores in Microsoft Visual Basic(V). The exercises are written assuming you are using Visual Basic 2005 and the .NET Framework version 2.0 (or later). Some older versions will work the same, but you may need to figure out some differences. It should work with newer version also.

### Module Prerequisites

It would be helpful to have completed the TSK and SEM modules for the DSP/BIOS operating system but it is not necessary if you are familiar with tasks and semaphores.

### Labratory

#### Part 1

- In this part you will make a project that will have two threads that will run at the same priority and print to the console. Examine how the threads at the same priority execute in a round-robin fashion.
- Start up Visual Studio and create a new Visual Basic project.
- Create two threads and start them up with the same priority, Lowest.
- In each thread make a continuous while loop and print the thread name in the loop. Your code can look like this.

While 1

```
    Console.WriteLine("Thread1")  
End While
```

## Part 2

- In this part you will make a project with three threads. The first two threads will write to a single array. The data that each thread writes will be different. The third thread will print the contents of the array. Without locking the resource the threads will not write to the array completely before the other thread writes to it also. The contents of the array will be a mix of data from each thread.
- Start up Visual Studio and create a new Visual Basic project.
- Create three threads and start them up with the same priority, Lowest.
- Make an integer array in the Form1 class. This should be just below the class statement.

`Dim IntArray(100) As Integer`

- In the first thread, make an infinite loop and within the loop write the numbers 1-100 (count) into the array. Have the thread sleep for 10 ms each loop too.

```
Dim count As Integer
While 1
  For count = 1 To 100
    Thread.Sleep(10)
    IntArray(count) = count
  Next
End While
```

- In the second thread, make an infinite loop and within the loop write the numbers `100-count` into the array. Have the thread sleep for 10 ms each loop too.
- In the third thread, have it pause for ten seconds or so (`Thread.Sleep(10000)`) and then print the values from the array. Do this one time.
- Run the program and observe whether the array values get written and printed in order.
- You may not notice but the printing could get interrupted by the other threads.

## Part 3

- In this part you will add a semaphore to the previous project so that the data gets written to the array without interruption. Also, the data will get printed without interruption.
- Start up Visual Studio and create a new Visual Basic project.
- Copy all the code from the previous part into this new project.
- To the code in the previous part, add a semaphore with a maximum value of 1 and an initial value of 1.
- For the threads that write to the array, add a `WaitOne()` call before the `for` loop that writes to the array and a `Release()` call after the `for` loop.
- For the thread that prints the array data, add a `WaitOne()` call before the printing of the array and a `Release()` call after the printing.
- Run the program and observe whether the array values get printed in order.

## Laboratory Write-up

- Print out your projects' code.
- Explain what is happening in each part.

## Microsoft Visual Basic Queues

This module describes the very basics of Microsoft Visual Basic queues and also the `AutoResetEvent` event.

### Introduction

This module describes the basics of Visual Basic queues and also describes the `AutoResetEvent` event.

### Reading

- Queue Class: <http://msdn.microsoft.com/en-us/library/system.collections.queue.aspx>
- Queue members: [http://msdn.microsoft.com/en-us/library/system.collections.queue\\_members.aspx](http://msdn.microsoft.com/en-us/library/system.collections.queue_members.aspx)
- Queue.Synchronized Method: <http://msdn.microsoft.com/en-us/library/system.collections.queue.synchronized.aspx>
- AutoResetEvent Class: <http://msdn.microsoft.com/en-us/library/system.threading.autoresetevent.aspx>

### Queues

Queues in Visual Basic (VB) are very simple to work with and very flexible. Defining a queue object is done with the following command:

```
Dim myQ As New Queue
```

This creates a single queue object named `myQ`. The main methods described in this module are:

```
myQ.Enqueue()  
myQ.Dequeue()
```

The **Enqueue** method puts items on the queue and the **Dequeue** method takes items off the queue. With these two methods the queue works as a first-in-first-out (FIFO) type of queue. A property of the queue object that is very useful is the count property:

`myQ.count`

This returns the number of items in the queue.

**VB queues are not thread safe.** This means that the operating system does not prevent multiple threads from modifying a queue at the same time. This can cause problems if you have multiple threads reading and writing to the same queue.

If you have multiple threads removing items from a queue you can run into problems. If a thread accesses the **count** property and it says there is an object on the queue and then the thread tries to access that element, another thread may remove it from the queue before the first thread has a chance to access it. If there is only one thread that removes items from a queue you will not have this problem.

To assure that a queue is thread safe you should use the **Synchronized** method of the queue. This method returns a wrapper for the queue that assures that the queue is thread safe. It does not fix the problem where two threads are taking items off the queue at the same time though. To use the **Synchronized** method first make a queue object and then make an object with the **Synchronized** method.

```
Dim myQ As New Queue
    Dim mySyncdQ As Queue =
Queue.Synchronized(myQ)
```

The queue **mySyncdQ** is a thread safe queue.

To synchronize threads that communicate using a queue, there is a need for the writing thread to inform the reading thread that there is something

available on the queue. This signaling prevents the reading thread from wasting time checking to see if anything is available on the queue. The reading thread will simply wait to be signaled that something is on the queue and then it will access the queue. To do this you can use an `AutoResetEvent` event.

From the MSDN help files: "`AutoResetEvent` allows threads to communicate with each other by signaling. Typically, this communication concerns a resource to which threads need exclusive access. A thread waits for a signal by calling `WaitOne` on the `AutoResetEvent`. If the `AutoResetEvent` is in the non-signaled state, the thread blocks, waiting for the thread that currently controls the resource to signal that the resource is available by calling `Set`. Calling `Set` signals `AutoResetEvent` to release a waiting thread. `AutoResetEvent` remains signaled until a single waiting thread is released, and then automatically returns to the non-signaled state. If no threads are waiting, the state remains signaled indefinitely. If a thread calls `WaitOne` while the `AutoResetEvent` is in the signaled state, the thread does not block. The `AutoResetEvent` releases the thread immediately and returns to the non-signaled state. There is no guarantee that every call to the `Set` method will release a thread. If two calls are too close together, so that the second call occurs before a thread has been released, only one thread is released. It is as if the second call did not happen. Also, if `Set` is called when there are no threads waiting and the `AutoResetEvent` is already signaled, the call has no effect."

To create an `AutoResetEvent` event use a command like:

```
Dim mySyncdQEvent As New AutoResetEvent(False)
```

This will create an `AutoResetEvent` event called `mySyncdQEvent` with the initial value of `False`. When your code writes to a queue you can use the `Set` method to signal to the reader thread that something is on the queue. The reader thread can use the `WaitOne` method to wait to be signaled. The code for the writer thread to put something on the queue and then signal the event is:

```
mySyncdQ.Enqueue(Something)
mySyncdQEvent.Set()
```

The reader thread can use code like:

```
mySyncdQEvent.WaitOne()
While mySyncdQ.Count
    Console.WriteLine(mySyncdQ.Dequeue())
End While
```

This will wait until something is available on the queue and then take all the items in the queue off and print the contents. This assumes that there is **only one reader** for the queue. If you have more than one reader for the queue you will need to lock the resource before using it. This is not explained here.

## Example

This example starts with a VB Form application that contains one TextBox named **TextBox1**. Double clicking on the TextBox will open the code view and make a **TextBox1\_TextChanged** method. As you type into the TextBox this method will be executed for each character that changes.

The example has one queue called **myQ** and is used as a synchronized queue with **mySyncdQ**. There is also a thread named **thread1** that is started that runs the method **ThreadMethodReceive**. This method will remove items from the queue and print them to the console.

Every time the text changes in the TextBox the **TextBox1\_TextChanged** method puts the contents of the TextBox onto the queue and signals a **AutoResetEvent** named **mySyncdQEvent**. The thread **thread1** calls the method **WaitOne** for the **AutoResetEvent** event. This will cause the thread to block until there is something on the queue.



```

Imports System.Threading
Public Class Form1
    ' Thread that will receive the messages that
are put on the queue
    Dim thread1 As New Thread(AddressOf
ThreadMethodReceive)
    ' Queue where message will be stored
    Dim myQ As New Queue
    ' Thread safe queue that will be used
    Dim mySyncdQ As Queue =
Queue.Synchronized(myQ)
    ' Event used to synchronize the TextBox putting
items on the queue and the thread
    Dim mySyncdQEvent As New AutoResetEvent(False)

    ' Method executed when the form is loaded
    Private Sub Form1_Load(ByVal sender As
System.Object, ByVal e As System.EventArgs)
Handles MyBase.Load

        ' Make this a background thread so it
automatically
        ' aborts when the main program stops.
thread1.IsBackground = True
        ' Set the thread priority to lowest
thread1.Priority = ThreadPriority.Lowest
        ' Start the thread
thread1.Start()
    End Sub

    ' Thread that runs separate from the Form1
thread
    Private Sub ThreadMethodReceive()
        ' Loop forever
        While 1
            ' Half second pause before checking the sync
event

```

```

        Thread.Sleep(500)
        ' Wait on the sync event. If something has
been put on
        ' the queue there will be an event.
        mySyncdQEvent.WaitOne()
        ' Continue removing items from the queue until
it is empty
        While mySyncdQ.Count
        ' Print the items removed from the queue
        Console.WriteLine(mySyncdQ.Dequeue())
        End While

    End While

End Sub

Private Sub TextBox1_TextChanged(ByVal sender
As System.Object, ByVal e As System.EventArgs)
Handles TextBox1.TextChanged
    ' Put the content of the text box on the queue
    mySyncdQ.Enqueue(TextBox1.Text)
    ' Signal that something has been added to the
queue.
    mySyncdQEvent.Set()

End Sub
End Class

```

## Microsoft Visual Basic Delegates

This module describes the very basics of delegates in Microsoft Visual Basic.

### Introduction

When developing Visual Basic applications in a multithreaded environment it may be necessary for a thread to access a GUI object in the form thread. However it is not possible for the thread to access the object directly so it must use a delegate that runs in the thread that handles the GUI object. This module describes the very basics of how to use delegate methods so that more than one thread may access a GUI object.

### Reading

- Delegate Class: <http://msdn.microsoft.com/en-us/library/system.delegate.aspx>
- Control Class: <http://msdn.microsoft.com/en-us/library/system.windows.forms.control.aspx>
- Control Class Invoke method: <http://msdn.microsoft.com/en-us/library/zyzhdc6b.aspx>

### Background

When a simple Visual Basic (VB) GUI application is made, there is a form class that handles all the GUI controls. This form class is a thread that handles all the interaction with the controls. It is possible to have other threads in our application. However, it is not possible for the other threads to have direct access to the GUI controls. The reason for this is it could be possible for the form thread and the other thread to try and change the same control at the same time.

To fix the problem of accessing the same control from two threads, a delegate function is made in the thread that handles the control object. The delegate function is invoked by other threads and the delegate function operates in the thread that handles the control object.

From the MSDN: "Delegates are similar to function pointers in C or C++ languages. Delegates encapsulate a reference to a method inside a delegate object. The delegate object can then be passed to code that calls the referenced method, and the method to be invoked can be unknown at compile time. Unlike function pointers in C or C++, delegates are object-oriented, type-safe, and more secure."

"The Delegate Class represents a delegate, which is a data structure that refers to a static method or to a class instance and an instance method of that class."

## Visual Basic Delegates

Suppose you have a TextBox control object in your Form called **TextBox1**. You can change the text in the TextBox by issuing a command like this:

```
TextBox1.Text = "New Text In Text Box"
```

This will change the Text property of the TextBox control object. When this is done within the Form thread everything works as expected. For instance, suppose you have a button in your Form called **Button1**. Then in the click method for that button you could have something like this:

```
Private Sub Button1_Click(ByVal sender As  
System.Object, ByVal e As System.EventArgs)  
Handles Button1.Click  
    TextBox1.Text = "I just got clicked"  
End Sub
```

This would all take place within the Form thread. The Form thread is the thread that created the TextBox control and the Button control so this is all within the same thread.

Suppose now you have a thread that you started that runs separate from the Form thread. If the thread tries to change the TextBox object you will get an error. If the thread looked like the following code, there would be a problem.

```
Private Sub Task1()  
    TextBox1.Text = "Task Text"  
End Sub
```

The problem occurs because the thread Task1 is running separate from the Form thread so the Form thread and the Task1 thread could change the TextBox object at the same time.

To fix this problem you can use a Delegate that points to a function that runs in the Form thread. Suppose the code was changed to the following:

```
' This defines a delegate subroutine that takes an  
integer as its input  
    Delegate Sub InvokeDelegate(ByVal myStr As  
String)  
    ' This makes an instance of the delegate that  
points to the address  
    ' of the InvokeMethod function. When the  
delegate is invoked it will  
    ' run the InvokeMethod method on the thread  
that owns the window handle.  
    Dim IDel As New InvokeDelegate(AddressOf  
InvokeMethod)  
  
    Private Sub Task1()  
        ' The Invoke method executes the specified  
        ' delegate on the thread that owns the  
control's  
        ' underlying window handle  
        TextBox1.Invoke(IDel, "Task Text")  
    End Sub
```

```
Public Sub InvokeMethod(ByVal myStr As String)
    TextBox1.Text = myStr
End Sub
```

This code section does not contain the whole program since it doesn't show starting the Task1 thread, etc. Lets look at the code one line at a time.

```
Delegate Sub InvokeDelegate(ByVal myStr As String)
```

This line defines a delegate that will have one parameter on its input and that parameter will be passed by value and is a string. The delegate type is called **InvokeDelegate**. The name of the delegate type can be anything.

```
Dim IDel As New InvokeDelegate(AddressOf
InvokeMethod)
```

This line creates an instance of the delegate type defined above (**InvokeDelegate**). The method or function it uses for the instance is the static method **InvokeMethod** which is defined later in the code. This means that **IDel** is an instance of a delegate and the instance is a static method defined by **InvokeMethod**. When **IDel** is used it will execute an **InvokeMethod** method.

```
TextBox1.Invoke(IDel, "Task Text")
```

Control objects have a method called **Invoke**. This method will execute a delegate on the thread that created the object. So, we first create a delegate **IDel** and then when the **Invoke method** is called for the TextBox1 object, the **IDel** delegate gets executed on the thread that created the TextBox1 control. This thread happens to be the Form thread. The **Invoke** method will not continue until the delegate completes.

If you don't want the calling thread to wait for the delegate to finish you can use the control method **BeginInvoke**.

```
Public Sub InvokeMethod(ByVal myStr As String)
    TextBox1.Text = myStr
End Sub
```

This is the method that is used for the delegate. Notice that it has the same input as defined above (a string input passed by value). The method simply takes the input string and assigns it to the `TextBox1.Text` property.

## Example

In this simple example there will be a Form that contains a TextBox. Also, a thread will be started that periodically writes to the TextBox. The TextBox is created by the Form thread and while the Form is running you can enter text into the TextBox. However, periodically the other thread will write to the TextBox and clear anything that has been typed into it.

Since the thread is separate from the Form thread, there needs to be a delegate that the thread uses to write to the TextBox.

```
Imports System.Threading
```

```
Public Class Form1
    ' This defines the thread that will run method
    ThreadMethod
    Dim thread As New Thread(AddressOf
    ThreadMethod)
    ' This defines a delegate type that takes an
    integer as its input.
    ' The delegate type is named DelegateType.
    Delegate Sub DelegateType(ByVal myStr As
    String)
    ' This makes an instance of the delegate type
    that points to the address
    ' of the ThreadMethod method. When the
    delegate is invoked it will
    ' run the InvokeMethod method on the thread
```

that owns the window handle.

```
Dim IDel As New DelegateType(AddressOf  
DelegateMethod)
```

```
Private Sub Form1_Load(ByVal sender As  
System.Object, ByVal e As System.EventArgs)  
Handles MyBase.Load
```

```
    ' When the form loads the other thread will be  
    set up and started
```

```
    ' Make this thread a background thread so it  
    automatically
```

```
    ' aborts when the main program stops.
```

```
    thread.IsBackground = True
```

```
    ' Set the thread priority to lowest
```

```
    thread.Priority = ThreadPriority.Lowest
```

```
    ' Start the thread
```

```
    thread.Start()
```

```
End Sub
```

```
    ' This method is the thread that runs separate  
    from the Form1 thread.
```

```
    ' It continues to run and every 5 seconds it  
    invokes the delegate that
```

```
    ' runs in the Form1 thread.
```

```
Private Sub ThreadMethod()
```

```
    ' The Invoke method executes the specified
```

```
    ' delegate on the thread that owns the  
control's
```

```
    ' underlying window handle
```

```
While 1
```

```
Thread.Sleep(5000)
```

```
TextBox1.Invoke(IDel, "Task Text")
```

```
End While
```

```
End Sub
```

```
    ' This method runs in the Form1 thread
```

```
Public Sub DelegateMethod(ByVal myStr As
```



```
String)  
    TextBox1.Text = myStr  
End Sub  
End Class
```

## Microsoft Visual Basic Queue Lab

The module is a lab assignment to help you better understand the very basics of Microsoft Visual Basic queues.

### Introduction

This lab module will help you become familiar with the basics of queues in Microsoft Visual Basic. The exercises are written assuming you are using Visual Basic 2005 and the .NET Framework version 2.0 (or later). Some older versions will work the same, but you may need to figure out some differences. It should work with newer version also.

### Module Prerequisites

You should complete the Visual Basic Threads and Semaphores lab before this one.

### Labratory

#### Part 1

- In this part you will just put some items on a queue and then take them off and print them.
- Start up Visual Studio and create a new Visual Basic project.
- In the Form1 class make a queue variable and then a synchronized queue.
- In the **Form1\_Load** method put the following objects on the queue:
  1. The string "Hello"
  2. The string "World"
  3. The number 23
  4. The string "Twenty Three"
- Now make a for loop that iterates using the **Count** property of the queue.

- Inside the loop, dequeue items from the queue and print them to the console.
- Run your program and record the results.

## Part 2

- In this part you will create two threads where one thread will write to a queue and the other will take the items off and print them. No synchronization will be done in this part. The reading thread will need to keep checking to see if there is anything on the queue.
- Start up Visual Studio and create a new Visual Basic project.
- In the Form1 class make a queue variable and then a synchronized queue. Use the synchronized queue.
- Make two threads in the Form1 class and start them up in the `Form1_Load` method.
- In the method for the first thread:
  1. make a counter variable
  2. make a loop that loops forever
  3. in the loop increment the counter
  4. in the loop put the counter value on the queue
- In the method for the second thread:
  1. make a loop that loops forever
  2. in the loop make an if statement that checks the count property of the queue
  3. if the count value is not zero, take an element off the queue and print it
  4. if the count value is zero, print a statement that says there was nothing on the queue
- Run your program and record the results.

## Part 3

- In this part you will create two threads where one thread will write to a queue and the other will take the items off and print them.  
Synchronization will be done in this part. The reading thread will wait until there is something on the queue then take all the elements off and print them. The writing thread will signal the reading thread when it puts something on the queue.
- Start up Visual Studio and create a new Visual Basic project.
- In the Form1 class make a queue variable and then a synchronized queue. Use the synchronized queue.
- In the Form1 class make an **AutoResetEvent** event variable that will be used to synchronize the two threads.
- Make two threads in the Form1 class and start them up in the **Form1\_Load** method.
- In the method for the first thread:
  1. make a counter variable
  2. make a loop that loops forever
  3. in the loop: increment the counter, put the counter value on the queue, print a statement telling what was written, signal the **AutoResetEvent** event with the **Set** method
- In the method for the second thread:
  1. make a loop that loops forever
  2. in the loop print a statement saying that the task is about to wait
  3. have the thread wait for the **AutoResetEvent** event by calling the **WaitOne** method
  4. after the **WaitOne** call, make another while loop that keeps removing items from the queue as long as the count is greater than zero
  5. print the value of the element taken off the queue to the console
- Run your program and record the results.

## Part 4

- This uses the code from the previous part.

- After writing the value to the queue, have the first thread sleep for 100 ms using the command

`Thread.Sleep(100)`

- Run your program and record the results.

## Microsoft Visual Basic Delegate Lab

The module is a lab assignment to help you better understand the very basics of Microsoft Visual Basic delegates.

### Introduction

This lab module will help you become familiar with the basics of delegates in Microsoft Visual Basic(V). The exercises are written assuming you are using Visual Basic 2005 and the .NET Framework version 2.0 (or later). Some older versions will work the same, but you may need to figure out some differences. It should work with newer version also.

### Module Prerequisites

You should complete the Visual Basic Threads and Semaphores lab before this one.

### Labratory

#### Part 1

- In this part you will make a project with a thread that will try to change a TextBox text without using a delegate method. When the thread tries to change the TextBox text it won't work because the TextBox control is not handled by the thread.
- Start up Visual Studio and create a new Visual Basic project.
- On the GUI, put a new TextBox. By default it should have the name **TextBox1**.
- Create a thread called **Task1** and start the thread in the **Form1\_Load** method.
- In the thread method **Task1**, assign a string to the TextBox.
- Run your program and observe what happens. You should get an error when the thread tries to access the TextBox control. Record what the error says.

## Part 2

- In this part we will make a delegate that will allow the thread to change the text in the TextBox.
- Either start a new project or modify the one from the previous part. What you should have in your project is:
  1. A TextBox on the GUI with the name `TextBox1`.
  2. A thread called `Task1` that is started in the `Form1_Load` method.
- Add commands to define a delegate type and an instance of a delegate. If you want to use the names shown here you can. These statements should be right under the beginning of the Form1 class (right after "`Public Class Form1`").

```
Delegate Sub DelegateType(ByVal myStr As String)
    Dim IDel As New DelegateType(AddressOf
    DelegateMethod)
```

- Make a new `DelegateMethod` method that takes a string as an input and uses it to change the text of the TextBox.
- In the `Task1` method, use the `TextBox1` to invoke the delegate method. Have it write the text `"Task Text"` to the TextBox.
- Run your program and observe that the text `"Task Text"` gets written to the TextBox.

## Part 3

- In this part we will make a simple clock. All we will do is continuously get the current time and write it to the TextBox.
- In this part we will have everything that is in the previous part so you can start where it left off. Your project should have:
  1. A TextBox on the GUI with the name `TextBox1`.

2. A thread that uses the **Task1** method that is started in the **Form1\_Load** method.
  3. Commands to define a delegate and an instance of a delegate.
  4. A delegate method that writes the input string to the TextBox.
- In the thread, we will have it get the current date/time and then write it to the TextBox.
  - Put the following commands that will define objects to get the current date/time.

```
Dim count As Integer
    Dim dispDt As DateTime
    Dim datePatt As String = "M/d/yyyy hh:mm:ss
tt"
    Dim dtString As String
```

- In the thread make a continuous while loop that puts the date into the string **dtString** and then sends that string to the delegate. To get the date use the command:

```
dispDt = DateTime.Now ' get the time now
    dtString = dispDt.ToString(datePatt) ' Convert
the date to a string
```

- In the while loop make the thread sleep for 100 ms so that it is not doing too much processing. It will get the time/date and write it to the TextBox every 100 ms.
- Run your program and observe that the text displays the date/time and that the seconds change.